Open Source Development – From Software to Space

Submittal is partial fulfillment of the requirements of

SPSM 6000 Practical Research In Space Ops

Webster University

by Matthew Maier

to William Hoffman

11DEC2012

Release of Intellectual Property Rights

I, the undersigned, Webster University Graduate Student hereby release all intellectual property rights of my research and deliverables as submitted in partial fulfillment of the requirements of the Webster University SPSM 6000 course. Webster University may reproduce, modify, edit, distribute in printed or electronic version all, or portions, of this work for academic purposes without fee or encumbrances into perpetuity.

I release all claims to immediate and future rights of this material into perpetuity. I hereby testify by witness thereof my signature that this work is my original research and it contains only by reference or citation, other works in printed or electronic form, as accepted in academic research.

____
Typed SPSM Graduate Student

As witnessed by me on this __ day of _________ 2012.

____
Witnessed:

William C. Hoffman, Jr.
Director -Space Programs
Webster University
5475 Tech Center Drive, Suite 110
Colorado Springs, 80919, CO
719-590-7340
719-590-7343 Fax
hoffmanwc@worldnet.att.net

Abstract

If the use of open source licenses in one area of technology, like rapid prototyping, can produce significant cost savings, then a more open approach can produce similar cost savings in another area, like space. Open source is an approach to intellectual property (IP) which uses the law to guarantee access to IP rather than forbid access. SPSM 5900 Space Commercialization will address the implications of open source technology on commercial endeavors. SPSM 5600 Space Acquisition Law will be used to investigate the relationship between the government and open source. SPSM 5650 Space System Contracting sheds light on what open source means with respect to contracts. SPSM 5750 Space Systems Engineering will delve into how open source changes the design of complex systems. SPSM 5770 Space Operations Management takes a look at some of the changes to system and mission design that are implied by open source.

Keywords: space, open, source, technology, intellectual property, law, cost

Executive Summary

This paper lays out the results of my research attempting to answer the hypothesis: If the use of open source licenses in one area of technology, like rapid prototyping, can produce significant cost savings, then a more open approach can produce similar cost savings in another area, like space.

My research was filtered through five class subjects from my Master's program: SPSM 5600 – Space Systems Acquisition Law, SPSM – 5650 Space Systems Contracting, SPSM – 5750 Space Systems Engineering, SPSM 5770 – Space Operations Management, and SPSM 5900 – Space Commercialization.

Unfortunately, I do not feel I was able to directly answer the hypothesis. What I was able to do was support the following three major points, which make a strong analogous argument for an affirmative answer to the hypothesis. First, open source is a relatively new cultural and legal model for organizing effort around the top priority of solving technical problems. Second, open source software has proven itself successful based on principles that are starting to be applied to hardware, which demonstrates that the only relevant difference between the two is the amount of resources necessary to transition between design and utilization of the technology. Third, space technologies, like most other technologies, should be able to manifest the benefits of utilizing the open source development process.

Based on my research, I believe that the following questions remain to be answered more empirically. First, what is the relationship between the cost of open source hardware projects and their proprietary equivalents? Second, what is the legal status of hardware-specific open source licenses? Third, Is there a difference between the motivations of open source software and hardware developers?

Sen, R., Subramaniam, C., & Nelson, M. L. (2009). Determinants of the Choice of Open Source Software License. *Journal of Management Information Systems*, 25, 207-239. doi: 10.2753/MIS0742-1222250306

Introduction

My intention was to research the academic literature for peer-reviewed information on the savings that open source hardware could produce in space technology. In my limited search period (the class is nine weeks long) I found no research on the subject outside of a RepRap developer, which was the project that originally inspired the search. What I did find was strong arguments for further research on the cost savings of open source hardware and for applying the open source development philosophy to space systems. I was able to find far more research on open source software than hardware, and nothing peer-reviewed on space technology in particular, so this paper is heavily conceptual and relies on analogies drawn from well-documented open source software and poorly-documented hardware projects.

The ten articles in this paper will be used to support several points which collectively address my hypothesis. First, that open source is first and foremost a process for solving technical problems, which is best explained by starting before free software and working up to the current legal situation. Second, that the only substantial difference between open source software and hardware is in the scale of resources necessary to transition from design to implementation. Third, that space technology should be able to enjoy substantial benefits by adopting the open source development paradigm. Each of these major points is supported by sub-points that will be drawn out of these ten articles either directly or by analysis. The appendix includes a quick-reference for the conceptual structure that is developed over the following pages.

Summary

Open source means that the raw design files are intentionally made available to anyone who wants them. This is done via one of several variations on a unique kind of license. Since the license will not change throughout the life of the project, it is up to the original developer to choose the license terms that will best fulfill their goals over a potentially long period of time (pg. 208). Sen's paper examines the motivation(s) of the developer(s) and which strength of open source license they should choose based on their priorities.

If any organization can claim to define what open source means in software it would be the Open Source Initiative (OSI), a non-profit that reviews and approves licenses (p. 209). For OSI to approve a software license as open source it must guarantee free redistribution, unrestricted access to the source code, that derivative works will be released under the same license terms, and that anyone can use the software for anything that does not violate the license. Some other guidelines are that the license must not apply to other software used along-side the open source software and that it must not depend on any particular technology.

All of that is important, but the aspect of an open source license that has the greatest impact on the developer's decision is the copyleft restriction. If “copyright” is using the law to prevent anyone from using something for free, then using the law to guarantee that anyone can use something for free would be the opposite, or “copyleft.” There are three main categories of copyleft: “strong,” which requires that anything based on the original work must carry the same strong-copyleft term; “weak,” which allows derivatives of the original to be licensed differently in certain situations; and “non,” which makes no restriction on the license terms of derivative works.

The restrictiveness of the copyleft term attracts developers in more-or-less intuitive ways.

Skilled developers looking for a challenge (intrinsic motivation) tend to avoid non-copyleft projects because a nonrestrictive license attracts too many other developers with whom they have to split the achievement (p. 214). On the other hand, developers looking for status (extrinsic motivation) tend to prefer non-copyleft licenses because their work gets in front of more eyeballs (p. 215). A third group, those motivated by social obligations, tends to avoid non-copyleft to prevent hijacking by for-profit entities (p. 229). It is worth mentioning a fourth group of developers who are motivated solely by the work itself and just do not care about how it is licensed at all (p. 229).

It is important to note that, while economic incentives always have an effect, it is the intrinsic factors (challenge and creativity) that have the greatest effect (p. 230). Also, of the extrinsic factors, status had the greatest effect. No single factor stands out as being significantly more important than any other, but economic incentives stand out as being significantly less important than incentives like challenge and status. An administrator can attract certain groups by choosing the appropriate copyleft restriction, but they should always advertise the opportunity for achievement and respect before explicit financial incentives (p. 231).

Analysis

Sen's research examines the motivations of open source developers and finds that they are most strongly motivated by intrinsic appreciation for a challenge. Also, that when they are extrinsically motivated, it is usually by status, which is the community's recognition that they are able to solve challenging problems. Financial or economic incentives are actually the least influential of all the motivations studied.

This is point 1.4 in the appendix and it supports the first major point that the top priority of open source is the solution of technical problems.

Transition

Open source is created by people, with particular motivations, but it is defined by law. The next article goes into depth on the structure of the relevant licenses.

Cardona, Dr. J. J. G. (2007). Open Source, Free Software, and Contractual Issues. *Texas Intellectual Property Law Journal*, 15(2), 157-211.

Summary

The legal issues regarding open source software begin with the code that makes computers do things. “Source code” refers to the human-readable instructions written by the programmer. A compiler converts the source code into a machine-readable form: the object code. It is possible to decompile object code, but it might not work, if it does work the result is only functionally similar to the source code, is harder to edit, and probably violates copyright law. It is worth noting that there is an exception; it can be legal to decompile protected code to allow a new program to interface with the old one.

Back when computers pretty much all ran custom code, the source code was delivered along with the object code, or the source code was all that was delivered and the user compiled it on their own. As the distribution of code that could run on any computer increased, it required protection. During the 1980s and 90s the United States, Japan, the European Union, the World Trade Organization, and the World Intellectual Property Organization all agreed to protect computer programs under copyright law (p. 166). At the same time, companies stopped distributing source code and Richard Stallman founded the Free Software Foundation to coordinate the activities of people who agree that no one should have to pay for software. Stallman developed the GNU's Not Unix (GNU) operating system, Linus Torvald contributed the Linux kernel, and the result was an operating system (GNU/Linux) made entirely out of free software.

While free software is partially practical, it is primarily moral. According to Stallman, to be free software the users must have the freedom to run the program for any reason, access the source code, make and release changes, and redistribute copies at will. This is diametrically opposed to a proprietary philosophy which would label most of the aforementioned freedoms as piracy. In practice the moral goals of free software are only possible through the legal construct of copyright. At first glance, it would be simpler to just put everything in the public domain; for the author to deny or give away all copyrights. But this would allow the source code to immediately be distributed as proprietary object code. In fact, it would mean that if any change was made to the original, that the derivative work would be fully protected under the new author's copyright. Since free software's motivation is primarily moral, the principle has to be enforced. Thus, the GNU General Public License (GNU GPL) forces redistributors to apply the same license to any distribution, original or derivative, preserving all the freedoms of subsequent users.

Open source originated in the late 1990s when Netscape gave away the source code for its browser. Free software and open source software have extremely similar definitions; the important difference being how they view proprietary software and the effect that has on their copyleft clause. Free has labeled proprietary software as 'the enemy' and is generally considered idealistic. Open source considers the open development process compatible with proprietary requirements and has been characterized as pragmatic (p. 183). If the standard license for free software is the GNU GPL then the standard license for open source software is the Berkeley Software Distribution license (BSD) which, in contrast, does not include a copyleft clause. This means that free software cannot be converted into proprietary software, but open source software can. Open source would include any relevant license with or without a copyleft clause, but free software requires copyleft. Open source occupies the philosophical middle ground between the irreconcilable proprietary and free approaches.

It is the copyleft clause that causes issues. Because it imposes itself on all redistributions of the work, it is referred to as 'viral.' In fact, there has been debate over whether it should be called a license at all, since it seems more like a contract. This is an important distinction with significant legal ramifications. Copyright is an internationally harmonized body of law, but contract law varies considerably around the world, and even varies within a single country.

The details of the legal debate are beyond the scope of this paper. The important take-away is that contract law depends on privity, which means that a contract can only apply to the parties which originally entered into it. As much as an open source license might look like a contract, privity can only exist between the author and a direct licensee. There cannot be privity between an the author and any user who received the software via a license from a redistributor. A lack of privity means that contract law cannot be enforced. Therefore, the only situation in which contract law is even an option for interpreting copyleft licenses is when an author licenses directly to a redistributor. Copyright, on the other hand, follows the creation wherever it goes. Copyright will apply to all relationships between the author and other entities. Therefore, the author-user relationship is only covered by copyright. The author-redistributor relationship can be covered by both copyright and contract law.

Analysis

Cardona's chapter lays out the legal situation and attempts to clarify what law applies in different situations. Proprietary software, which prioritizes maximizing user costs, inspired a backlash of the exact opposite priority. Free software prioritizes minimizing user costs. It accomplished this goal by using copyright, previously the domain of proprietary interests, in reverse. Licensors have always required licensees to obey certain rules, free software simply rewrote those rules to prevent anyone from ever making the software proprietary, rather than to keep the software proprietary. Since these two cultures are formed around diametrically opposed priorities, they will forever be at odds with each other. Open source emerged as a compromise between the two. By prioritizing something other than cost, open source allows proprietary interests and free interests to compromise on a common interest, rather than remain in stalemate.

Because free software broke new ground when it inverted copyright and used it for the opposite of its intended purpose, questions remain as to its legal standing. Based on well understood aspects of copyright and contract law, there are two important relationships: the author-redistributor and the author-user. The former can fall under both copyright and contract law, but the latter can only fall under copyright law.

This is points 1.1, 1.2 and 1.3 in the appendix and it supports the first major point that the top priority of open source is the solution of technical problems.

Transition

Given the theoretical argument for how to interpret copyleft licenses, the next article will examine an actual court case hinging on how to interpret copyleft.

Hogle, S. (2008). Update: *Jacobsen v. Katzer* Represents a Major Victory for Open Source. *The Computer & Internet Lawyer*, 25(10), 1-3.

Summary

The most distinctive facet of open source is the copyleft clause, which utilizes copyright law to guarantee access to intellectual property rather than deny access. Part of this distinctiveness is the question of whether or not the copyleft clause is even enforceable in court and, if so, just how enforceable it is. If the copyleft clause can be interpreted as copyright then the plaintiff has the chance to get injunctive relief, which is when the court orders a party to do or not do something. On the other hand, if the copyleft clause is just a contract, the plaintiff will have to show actual damage and will only be able to recover enough to make the damage whole. Since open source licenses products for free, it would basically be impossible to show economic loss, rendering the copyleft clause effectively unenforceable. In practice, the only meaningful way for authors to enforce their copyleft license is to get the court to order the other party to stop or start some particular action, which can only happen under copyright.

The United States District Court for Northern California ruled precisely the opposite; that failure to comply with the portion of a license that requires attribution (naming the original author) was merely a breach of contract (p. 1). The court argued that the attribution requirement was not actually a condition of the license despite the fact that the language of the license allowed the licensee to reproduce the work 'provided that' attribution was clearly noted. The copyleft clause in the GNU GPL, which is the movement standard, is worded similarly. Thus the ruling cast doubt on whether or not the requirement to redistribute the work under the same license actually required redistributors to do anything.

On appeal, the Federal Circuit reversed the ruling of the lower court (p. 2). The Federal

Circuit determined that the license is granted only if the conditions are met. So, if someone downstream disregards one of the conditions, then they also lose the license, which puts them in violation of the author's copyright. Additionally, the Federal Circuit specifically upheld the goals and benefits of the free/open software movement and specified the need to preserve injunctive relief as a method of enforcing copyleft licenses.

Analysis

Hogle succinctly lays out an example of the ambiguity surrounding copyleft being resolved, and then re-resolved, in an actual court case. Ultimately the court interpreted copyright and contract law in such a way as to support the enforceability of copyleft under copyright law, which covers every relationship between the author and another party.

This is a good time to introduce the question of how patent law fits into the situation. When you encode an idea in a form it is protected by copyright, but the idea itself is not protected. Copyright allows for independent creation of the exact same artifact, both of which are protected; it has no uniqueness requirement. Patents, on the other hand, require novelty because they protect the idea itself. Even independent creation infringes on a patent[1].

Patent protection is even more extensive than copyright protection; there is no exception for fair use, no exception for non-commercial use, and merely using an infringing object is infringement, although you do have to infringe on the entire patent all at once. However, the idea that is protected is usually extremely specific, and most things are simply not covered by a patent at all. Additionally, you have to apply for a patent, which requires expense and disclosure. There have been attempts to expand copyright protection, which is issued the moment something is created, to functional objects, but they have never succeeded. It is possible that design patents

1 Weinberg, M. (2010). *It Will Be Awesome If They Don't Screw It Up*. Public Knowledge. Retrieved from http://publicknowledge.org/it-will-be-awesome-if-they-dont-screw-it-up

could be expanded until they start to fill that role, but they would still require active application.

This is point 2.3 in the appendix and it supports the second major point that developing open source software and hardware differ mostly in the scale of resources required. There do not seem to be new legal restrictions that appear when applying open source software principles to hardware. The designs of objects will still be covered by the same copyright law, but it is unlikely that the object itself will run afoul of patent law.

Transition

Since there is a strong argument that open source licenses have and will be upheld by the courts, and patent law will rarely apply, the next article examines a theoretical model which is based on the possibility that everything could become open source.

Maurer, S. M. (2012). The Penguin and the Cartel: Rethinking Antitrust and Innovation Policy for the Age of Commercial Open Source. *Utah Law Review*, 1, 269-318.

Summary

Open source used to be amateur, but now most of it is professional. Because open source is a good way to share costs it allows several companies to pool resources and produce the same code cheaper. However, since the whole group of them has the same basic code, none of them can offer better software. This creates a sort of cartel situation that has a negative effect on competition and investment.

In 2002 a sample of open source projects uncovered only 500, but as companies began paying their software engineers to contribute to open source projects, that number rose to 4,500 by 2006 (p. 271). Now most open source projects are intended to be bundled with proprietary or complimentary products. An interesting example is that IBM actually converted the Eclipse development environment from proprietary to open. When participation was disappointing, IBM gave up even more control by transferring Eclipse to an independent non-profit, induced an eventual 50 other companies to contribute code (p. 273). The business case for this is, more or less, as simple as the fact that if each company contributed a single software engineer, they only have to pay one person, but each company gets to benefit from software that was developed by 50 engineers. This arrangement has proven superior to traditional joint-ventures because things actually get done, rather than the previous situation of nothing getting done or the project being hijacked by one participant (p. 276).

The exact structure of Maurer's argument is beyond the scope of this paper. Suffice to say that, in an abstract market composed of only open source companies all sharing the same code base, their incentive to improve their code drops to zero because no company needs to worry about another company offering consumers a better product. Indeed, in a pure-open-source industry, anything that benefits one company benefits all equally, which means there is no competition. A lack of competition is effectively a monopoly. This means that open source is beneficial only in moderation.

Models of the situation suggest that the most total code is produced when 20% of the companies are open source and the rest are proprietary (p. 284). That also happens to be approximately the ratio that the Department of Justice uses to evaluate the danger of monopolization (p. 285). Models also indicate, and are backed up by observation, that in practice 60% of companies will practice open source, although they will tend to be small so that proprietary software actually serves the majority of consumers (p. 287). There are exceptions to the generic cases. For example, if consumers buy based on loyalty, then additional damage from cartelization could be effectively irrelevant.

Judges use section 1 of the Sherman Act to determine whether or not an agreement has relevant monopolistic effects. The basic reasoning tends to indicate a need to worry if 1) innovations that would have been justified in a more competitive situation are not pursued, 2) the end result would have been achieved faster outside of the agreement, or 3) the members would have pursued goals significantly more useful to the public. Real situations are complicated, but one guideline is that infrastructure objects like operating systems are probably not at risk of damaging cartelization. Another "safe harbor" is that any general market is probably not at risk of cartelization if only 20% of the industry uses open source.

An interesting implication is that copyleft, also referred to as a "viral license," encourages companies to join the open source group which reinforces cartelization. This means that, in addition to the financial incentives, there is a legal incentive or requirement to switch to open source. If the courts decide that a particular license is too broad, they can force the steward of the license to amend it. If there is no steward, they can simply declare the license in violation of anti-trust laws, which will create a strong defense for anyone who does not follow the copyleft provision in the license.

Maurer points out that, while strong copyleft terms can encourage a cartel, some amount of copyleft is necessary to preserve open source. He argues that the evidence from actual implementation of the range of copyleft strengths suggests that strong-copyleft, such as the GPL, is unnecessarily restrictive. Not only have the weaker versions been used extensively without corrupting the project, but the strong-copyleft aspect of the GPL was developed with the express intention of having the largest possible impact. Based on the success of several other (weaker) licenses, it is logical to conclude that it only takes moderate copyleft restrictions to stabilize an open source project. In fact, moderate copyleft licenses are by far the most popular (p. 309).

This popularity evolved over time. At first, the thinking was that if free is good, then more free is better. However, since open source volunteers are not strongly influenced by economic considerations, no one could come up with a feasible government incentive such as a tax break. The approach now is to simply look for business models that are the most successful, whether proprietary or open, and buy from them. One area in which government assistance of open source makes sense is to help avoid or remedy an all-proprietary market so as to gain the benefit of mixed open/proprietary competition. One thing governments can do is ensure the presence of open source bids in competition with bids from proprietary companies and then let the existing decision structure of “best value” direct the money. This would help open source penetrate proprietary markets without distorting market signals.

Analysis

An enlightening example of the way that open source is a successful, mature approach that sits between anti-profit and for-profit activities comes in the form of IBM. In 2005 IBM issued the “IBM Statement of Non-Assertion of Named Patents Against OSS” in which it named 500 patents it held and promised never to assert them against an open source software project[2]. In 2010, however, IBM asserted two of those patents against a decade-old and expanding open source software project that was threatening their extremely profitable mainframe business[3]. This is the same IBM that created the Eclipse software development environment and eventually gave it to an independent foundation[4]. So, rather than being a distinctly non-profit or for-profit activity, open source is sometimes compatible with commercial interests and sometimes opposed (or at least perceived that way). This is points 1.1 and 1.2 in the appendix and supports the first major point that open source does not focus on cost/price.

In his analysis of cartelization, Maurer starts from the existence of copyleft and extrapolates, using a simplified model, to the conclusion that open source is only beneficial in moderation. He makes an exception for objects like operating systems, which are basic standards that are shared widely across the industry.

This is point 3.1 in the appendix and it supports the third major point that open source should be able to produce benefits in the space domain. All industries need shared standards and it makes sense for them to be produced and maintained via open collaborations, and it seems unlikely that open source technology would be able to significantly replace well-established proprietary competitors.

2 http://www.ibm.com/ibm/licensing/patents/pledgedpatents.pdf

3 http://www.scribd.com/doc/29469085/IBM-letter-dated-11-March-2010-to-TurboHercules-SAS

4 http://www.eclipse.org/org/

Transition

If open source really can cause cartelization when deployed in excess, then it would be up to the government to oversee the balance. The next article investigates the government's relationship with open source licenses.

Tiefer, C., & Shook, W. A. (2012). Intellectual Property. In C. Tiefer, & W. Shook (Eds.), *Government Contract Law in the Twenty-First Century* (pp. 291-330). Durham, NC: Carolina Academic Press.

Summary

Intellectual property (IP) can be a sticking point between contractors and the government. The government is more inclined to want to supply relevant IP to any and all parties so as to promote open competition, which most likely means lower prices and fewer tax dollars expended. Contractors, on the other hand, are inclined to keep IP exclusive, which leads to greater profits.

Traditionally, the government took title to any inventions developed under contract, but that interfered with the use and improvement of promising technology. The Bayh-Dole act, and Presidential memoranda, allow contractors to retain the title to their inventions, under the assumption that this policy will best incentivize development of the invention. In return for the patent rights, the government obtains a nonexclusive, nontransferable, irrevocable, paid-up license to use the invention. It is important to point out that this policy grants the government rights to any invention "reduced to practice" during the contract, even if it was "conceived of" prior to the contract, and any invention "conceived of" during the contract, even if it was "reduced to practice" after the contract.

A similar set of rights is granted to the government when work done under a contract is copyrighted, including the right to reproduce, prepare derivative works, distribute copies, and perform or publicly display the copyrighted work. The same policy applies to copyrighted data not produced under the contract, but delivered under the contract.

While there is always the potential to clash over a specific issue, patents and copyright are well defined and have well-developed Federal Acquisition Regulation (FAR) clauses.

Analysis

Seeing as how open source only exists because the law supports it, and the government has a lot to say regarding the structure of the law, and the government is responsible for overseeing the balance between open source and proprietary competitors, it is important to understand the government's policies on intellectual property rights in general and open source specifically.

The FAR part 27 deals with Patents and Copyright[5]. Even one chapter in the FAR is too large to be fully analyzed in this paper, but some interesting notes can be drawn out of it. Not surprisingly, the government can only be sued for financial damage resulting from patent or copyright infringement; it is not possible to obtain injunctive relief (27.201-1(a)). It is possible for the government to use a patent without consent from the owner, particularly during a national emergency (27.204-1(c)). In 27.405-3(a) the FAR states that commercial software acquired by the government should use the same license that is used when the software is supplied to the public. However, the government has security and legal requirements that it cannot violate. Licenses need to be reviewed by a contracting officer to avoid situations like, for example, the government being required to provide its modified source code when the modifications are classified. There is little doubt that security would take priority over the copyleft clause, so it is better to avoid such contradictions proactively. The FAR part 52.227 also addresses commercial software.

Several publications have been released that specifically address questions regarding open source software and government acquisition activities. Again, the full content of the document is outside the scope of this one, but some important points can be drawn out[6]. The Defense

5 FAR Part 27. Retrieved from http://acquisition.gov/far/current/html/FARTOCP27.html

6 Michel, B. S. et al. (2011). *Government Computer Software Acquisition and the GNU General Public License*. Software Freedom Law Center. Retrieved from

Federal Acquisition Regulation Supplement (DFARS) also considers open source software, even strong-copyleft like the GNU GPL, to be commercial (pg. 8). The GNU GPL license does not require that modifications be returned to the original developers or released to the public; they can be maintained internally, which fits nicely with the government's need to protect classified modifications (p. 4). This is not interpretation; it is actually the official stated position of the Free Software Foundation (author of the GPL) (p. 12).

As another example, in a 16 page document released in 2010, the Under Secretary of Defense for Acquisition, Technology, and Logistics outlining in explicit detail 23 principles to improve federal acquisition results. Some principles required a full page of explanation. Requiring that open alternatives be compared to closed systems architectures, however, was merely stated in a single paragraph and required no supporting explanation[7].

Additionally, open source software is so successful in infrastructure missions that it has simply become hard to avoid. Many phones run the Android operating system, and the internet runs on the open source “LAMP stack” consisting of Linux (operating system), Apache (server), mySQL (database) and Perl (programming) (p. 3). This supports point 3.2 in the appendix because it is open source penetration of a large market, which supports the third major point that even the space industry should be able to realize benefits from open source technology.

Not every government agency is required to use the FAR as it stands, and NASA is one of those exceptions. They have their own open source software license, approved by the Open Source Initiative (OSI), called the NASA Open Source Agreement (NOSA)[8]. NASA cites the numerous benefits of open development and the fact that their new open license helps them comply

http://www.softwarefreedom.org/resources/2011/government-acquisition-and-gpl.pdf

7 http://www.acq.osd.mil/docs/USD_ATL_Guidance_Memo_September_14_2010_FINAL.PDF

with the Open Government initiative.

A guide published by the Office of the Staff Judge Advocate Space and Missile Systems Center specifically to lead program office personnel through the acquisition of software systems is a much more specific reference[9]. The closest it comes to making a distinction between acquiring software and acquiring hardware, or the "medium" that software resides on, is to mention that they are owned/licensed separately and the difference needs to be taken into account (p. 6).

This supports point 2.2 in the appendix which supports the second major point that the only substantial difference between open source software and hardware is the amount of resources required. Additionally, NASA's proactive creation of an open source license tailored to their specific situation, specifically to capture the benefits of open source development, supports point 3.3, which supports the third major point that space should be able to benefit from the open source development model.

Transition

The government can be supportive of open source, at least in moderation and/or specific circumstances, which parallels commercial entities. The next article looks at actual release of technology into a commercial market.

8 Herron. S. (2012). *NASA Open Source Software Development*. National Aeronautics and Space Administration. Retrieved from http://www.nasa.gov/open/plan/open-source-development.html

9 *Acquiring and Enforcing the Government's Rights in Technical Data and Computer Software Under Department of Defense Contracts: A Practical Handbook for Acquisition Professionals 3rd ed*. (2011). Office of the Staff Judge Advocate Space and Missile Systems Center. Retrieved from http://mil-oss.org/resources/us-airforce_acquiring-enforcing-governments-software-rights-under-dod-contracts.pdf

Chiesa, V., & Frattini, F. (2011). Commercializing Technological Innovation : Learning from Failures in High-Tech Markets. *Product Development & Management Association*, 28, 437-454. doi:10.1111/j.1540-5885.2011.00818.x

Summary

High-tech markets are particularly difficult to break into because they are volatile and interconnected. While there is a lot of research on why innovation works, or not, there is less regarding how the process of commercialization affects the success of a new technology. Chiesa's research compared several hardware innovations (Apple Newton, IBM PC-Junior, Tom Tom GO, Sony Walkman, 3DO Interactive Multiplayer, Sony MiniDisc, Palm Pilot and Nintendo NES) based on two dimensions: how much support they required from the community (systemic) and how succesful they were.

There was a lot of focusing of the research, from a large list of products to a small one, and from a large list of factors to a small list. Strategic decisions are those made before launch, or even before development. Tactical decisions are those made during and after launch; how to actually implement the strategy when the time comes. The most relevant factor for measuring successful commercialization was customer acceptance. Of the reasons for failure, lack of support in the network and negative attitudes from early adopters are salient.

Due to the growth of global markets, entities in high-tech industries are strongly dependent on each other's decisions. This has led to more modularity, unbundling, specialization and deverticalization. Entities will not adopt a new technology unless they think most other entities will as well. The decision depends on end users, suppliers of complimentary technology and entities that supply information on the technology, among others. However, this applies more to systemic innovations (lots of interdependencies) than to autonomous innovations (few interdependencies).

When entities consider adopting new technology they tend to ask early adopters for their opinion because without someone actually trying the technology there is too much risk of it not working for some reason. Thus, early adopters exert a strong influence on the market.

In a result that may or may not be surprising, Chiesa's research identified that inter-firm relationships are important for gaining the support a new innovation needs to be accepted by the network. Denying other entities the ability to incorporate the technology, and/or produce complementary technology, is a bad move. Even if the network welcomes the new technology with open arms, if it is impossible, or prohibitively difficult, to interact with the new technology then network acceptance will not occur. Whenever possible, a core aspect of the strategic commercialization plan should be to lower the barriers other entities have to jump over to adopt the technology. The article cites the 3DO Interactive Multiplayer's software development kit, which sold for thousands of dollars, as an example of what not to do, and Palm's development kit for the Pilot, which was released free of charge, as an example of what works much better (p. 448). Failed commercialization attempts were characterized by no partnerships, or traditional commercial transactions, whereas successful attempts tended to use long-term cost-sharing arrangements.

Delaying these partnerships until the innovation has gained acceptance in the marketplace is likely to result in failure. For example, Sony did not form arrangements with retailers that would encourage them to support the MiniDisc, assuming that when it became popular the support would come naturally (p. 449). But, without initial active support from important industry partners, the MiniDisc failed. The chicken-and-egg problem neatly describes the challenge of getting something to happen when it depends on a second thing that also depends on the first thing. However, high-tech markets have an idiosyncrasy in that early adopters tend to be attracted to the fact that something is new and consider its systemic support to be largely irrelevant (p.

449). The systemic support will be important to later customers, who tend to view a vibrant eco-system of complimentary products/services as a signal of the innovation's value.

A big part of developing network support and complementary technologies is to clearly describe what the new innovation does, and make sure it does that thing well. Early adopters are the gateway; if they judge the innovation to be broken in some way the negative reaction can prevent the innovation from being commercialized even if all of the problems are eventually fixed. It is important for the strategic plan to both target new technology as specifically as possible and limit features that do not add value for the core demographic.

Analysis

Chiesa's research focused on hardware technology like Sony Walkman, Palm Pilot and Apple Newton and on what strategic and tactical decisions were the best predictors of success, as measured by customer acceptance. The results demonstrate that modularity is important because, all other things being equal, a piece of technology that requires a lot of support will be less likely to find acceptance. Additionally, the chances of success can be maximized if all interested parties are included from the beginning, so that features can be clarified/targeted and cost-sharing arrangements can be made.

An illustration of how open source principles apply directly to hardware is the Arduino microcontroller. It was developed basically as a class project and was open sourced to allow for ongoing support. Because of its open source nature, it quickly became the standard microcontroller for artists and hackers who needed a microcontroller, but did not have the engineering experience to use the proprietary models or design their own. Since all of the relevant design files are freely available, the community was able to easily share what they learned, which meant that new users could find out everything they needed to know about the Arduino without actually having to buy one, and without expensive and time-consuming trial and error after they bought one. A thesis on the Arduino[10] devotes a whole eight lines of text to explaining that open source started with software. The extrapolation from open source software to hardware is distinguished only by being remarkably intuitive; even when the subject comes up it barely requires an explanation. In fact, several times in Gibb's thesis the Arduino is implied to be a natural extension of the Open Source Initiative (OSI) (p. 2, 10, 38, 40, 65), despite the self-stated mission of OSI being "…a development method for software…"[11].

This is point 2.1 in the appendix and supports the second major point that the principles of successful software and hardware projects are the same; the only significant difference being the amount of resources each requires. Additionally, this supports point 3.1 in the appendix because it further emphasizes that there is no indication of a limit to the technology that open source can help improve, although the greatest benefits seem to be in standardized infrastructure, which supports the third major point that space technology should be able to benefit from open source development.

Transition

The commercialization of hardware tends to be aided by the same sorts of things that are natural strengths of open source development. The next article goes into more depth on the strengths and weaknesses of open source.

---

10 Gibb, A. (2010). *New Media Art, Design, And The Arduino Microcontroller: A Malleable Tool* (Master's Thesis). Retrieved from http://c298394.r94.cf1.rackcdn.com/New%20Media%20Art,%20Design,%20and%20the%20Arduino%20Microcontroller.pdf

11 http://opensource.org/

Iqbal, J., Quadri, S. M. K., & Rasool, T. (2011). Open Source Systems and Engineering: Strengths, Weaknesses and Prospects. *Trends in Information Management*, Vol 7, No 2. Retrieved from http://www.inflibnet.ac.in/ojs/index.php/TRIM/article/viewFile/1253/1134

Summary

Open source is no longer a "fringe" activity; it has been accepted by commercial entities and has demonstrated its viability in legitimate competition with alternatives. Open source breaks away from traditional software engineering by producing a product via directed evolution rather than planned creation. Instead of keeping things in-house, all of the relevant work is made available on the internet for anyone who is interested.

There are several reason open source has been adopted by the software development industry: it is usually free, development and debugging are massively parallel, more people are usually involved, the total resources applied to the challenge is often greater, the quality usually remains high, and the developers share a common social bond and peer recognition.

In contrast to the traditional process, open source tends to blend several steps into one. For example, project requirements are largely assumed ahead of time and the structure is laid out by a core group of developers. Modularization is the key to making the subsequent development process parallel and distributed. It lowers the barriers for new and/or isolated developers to contribute because they have less to learn and their changes are less likely to negatively impact other developers.

Open source development, when compared to traditional software engineering, exhibits an array of strengths and weaknesses. One strength is release frequency or, rather, the potential for frequent releases. With numerous developers working on their own schedule all around the world is just makes sense to release the work early and often. One thing that always translates into any language, and at any time of day, is a bug. The software either works or it doesn't. Since their efforts are only loosely coupled, it is important that developers find out if all of their different modular pieces work together quickly so that they can address mistakes and misunderstandings. It also means that forks and new features become part of the dialogue rather than languish in obscurity. The horizontal organizational structure allows frequent releases to be a strength, not a requirement, of open source. It is entirely possible for open source software developers, particularly groups that have a better defined organization, to release better integrated software less frequently.

A related strength of open source lies in parallel development and debugging. Traditional software engineering has a problem with exceeding budgets, failure to meet deadlines, and customer dissatisfaction with the eventual result. A crisis can cripple or kill an otherwise promising project. Open source has a more resilient structure than the traditional model and is better able to handle a crisis. By not having fixed relationships, or at least leaving them loosely coupled, open source development projects can rearrange to handle problems. With a horizontal organization that has activities occurring in parallel, newly developed capabilities get tested immediately, or not. If resources need to shift to another aspect of the project they can. Additionally, when the software reaches the users (non-developers) they can contact the developers directly and quickly clarify, often through direct conversation, what is a bug and what is a feature. Customer feedback can be immediately incorporated into the parallel development activities.

In fact, the community of users and user-developers confers upon open source a strength that is the opposite of a cliché weakness of traditional software engineering: adding people to a late project makes it later. That can be true of design activities, and of rigidly hierarchical organizations, but it is not true of brain storming or quality assurance activities. Traditional software engineering has to go looking for tests that are extensive and rigorous enough to turn up unusual bugs. Open source has those tests built into its structure. The places that open source software ends up, and the missions it gets applied to, can vary widely almost as soon as the software is released. If you add more people to the project it gets better faster.

Those people will not just be contributing to the project, they will be learning along the way. In open source development the developer is usually accessible and willing to answer questions about their work. Additionally, writing up some kind of instruction or introduction is a good way to raise one's status in the community. When the work is all a new developer has access to, they benefit from the fact that the notes are not stripped out. Open source does not separate effort into work-silos and training-silos but combines them and provides an education as well as a product.

For the people who do not develop the project, but only use it, the level of input they can provide to the developers is unheard of in traditional software engineering. Sometimes users go directly to the core developers, sometimes the team sets up a bug management system, but there is usually fast, two-way communication. The users get to see the improvements quickly, and the developers have access to the user who reported the bug in case they have additional questions.

That being the case, open source is not without its weaknesses which, like its strengths, are directly linked to the horizontal, heterogeneous structure. The lack of a formal, hierarchical organization means that open source does not usually make systemic changes. Decisions affecting the entire project are usually made at the beginning and then never changed. If later it turns out the project needs to go in another direction, rather than go through some kind of traditional unfreeze-move-refreeze process, the project just forks. The fork creates a new "beginning" where systemic changes can be made and the state of the old project is used as raw materials for the "new" project. Projects that do not fork are often full of patches and dependencies which would be unnecessary from a system perspective.

A related weakness is that redundancy can increase far beyond what a traditionally managed process would allow. It is entirely possible, even expected, for different developers to solve the same problem in total ignorance of each other. The inverse of this weakness is the additional weakness that, because tasks are not handed down in a systematic way, the undesirable work might not get done at all. For the most part, developers choose what they want to work on, and they tend to choose the problems with the greatest associated challenge and/or prestige. Work that is not creative, hard, or dependent on extensive education, can be left unfinished.

The easy, boring work does, however, lead to the next weakness of open source. Because it is open to anyone who is interested, open source attracts a lot of newcomers. Even newcomers who are already technically proficient will need to learn about the specific project, but newcomer orientation is a job that few people who are knowledgeable about the project want to waste their time on. Odds are that the newcomer will not stick around long enough to do anything constructive, and if they do then they might want to take things in their own direction anyway. There are incentives to put a great deal of effort into peers, but the meritocracy of open source requires that someone first demonstrate their proficiency before they are considered a peer by the developers. The easy work that makes up in tedium what it lacks in prestige is left for the newcomers to do as a sort of self-education and culling process.

Something that can definitely interfere with the learning process, and with the rest of the development process for that matter, is the possibility of people living in different time zones and cultures. There are plenty of software engineers that have trouble communicating in English and not all projects are started by English-speaking developers. Even when there is an honest attempt at communication and support it can get lost behind misunderstandings.

Finally, due to the lack of formal structure and mechanisms, open source projects can easily become over-dependent on one or two key leaders. Keeping a decentralized project going, particularly one that grows large and complex, requires extensive knowledge of both the subject and the project combined with an unusual ability to inspire and manage peers. If key personnel become unavailable for one reason or another it might be impossible to fill their shoes, leading to failure of the project.

Analysis

The flourishing of open source, despite no real marketing or advertising, speaks to its prospects in the future. It continues to step into new domains and to be adopted by new partners. Its focus on modularity and community mean that it grows from the bottom-up without having to be imposed from the top-down. Rather than having mostly weaknesses or mostly strengths, open source is merely a different way to do things, which creates trade-offs.

This is point 2.1 in the appendix and supports the second major point that open source is scalable. It also supports point 1.4 in the appendix, which supports the first major point that open source prioritizes solutions.

Transition

Open source shows promise as a scalable technology development process. The next art-icle examines a specific open source project that is a unique application of principles developed in software being applied to hardware.

Bruijn, E., (2010). *On the viability of the open source development model for the design of physical objects Lessons learned from the RepRap project* (Master's thesis, University of Tilburg). Retrieved from http://www.scribd.com/doc/44555887/On-the-viability-of-the-open-source-development-model-for-the-design-of-physical-objects-Lessons-learned-from-the-RepRap-project

Summary

There has been a lot of research on open source software, but little-to-none on applying the same development process to physical objects (p. 2). While it is not the first open source hardware project, the RepRap project is the focus of Bruijn's thesis because he was, and still is, a core developer. The RepRap project is aimed at creating a cheap machine that can fabricate arbitrarily shaped physical objects and thereby reproduce all, or at least a significant percentage, of its own parts. Bruijn documents support for the conclusion that open source hardware development is viable and discusses generalization of his findings.

Something that is important in software, but even more important in hardware, is a pool of components that developers can build upon. Open software benefits from, but open hardware depends on, modular design. The ability to change one thing without affecting anything else is a key enabler of collaboration in open hardware. Without modularity, it is difficult to attract other developers to one trajectory. Additionally, while it will always cost money to physically build something, if there is a pool of modular components the expense (in time and money) of designing can be largely skipped. For example, the most common structural component of the RepRap is threaded rod, with matching nuts and washers, because it is so easy to adjust and rearrange. Another good example is that major subsystems, such as the extruder and the control electronics, can be easily switched out for completely different versions.

The RepRap starts with a digital file and a blank build surface, it then lays down noodles of plastic one layer at a time until it has built up a solid, tangible version of the digital file. This is called additive manufacturing, which is hardly new. The additive manufacturing industry has existed for decades but never produced any machines less than several tens of thousands of dollars. The RepRap, on the other hand, has existed for only a few years and costs less than a thousand dollars. Several small businesses have sprung up to sell RepRaps, or derivative 3D printers, and are already surpassing the sales volume of Stratasys Inc, which had led the additive manufacturing industry for 7 years.

There are other open hardware projects that have been remarkably successful, but it is difficult to think of any that revolved around an actual machine. For most people, even open source developers, “hardware” means electronics. A big part of RepRap's position as the first machine to replicate the success of software projects is probably that it is a tool for translating digital files into physical objects. Distributing new ideas via the internet is a core competency of open source development, and it is a lot easier to capture the benefit from that when you can have the machine do the fabrication work for you. A popular anecdote from Bre Pettis, one of the founders of Makerbot (a RepRap derivative), is that he exchanged improvements to a whistle with someone in the Netherlands in less than an hour, which is far faster than a whistle could have been shipped overseas. This new capability allows the community to much more closely parallel the incentive structure of open source software. In fact, reports from RepRap developers indicate that being in close physical proximity is unnecessary, although it can be helpful.

Another way in which the RepRap project parallels open source software development is that the tools are as much a part of the project as the final result. Arguably, in the case of RepRap the tool is the final result, but the RepRap is still more of a tool because in addition to making its own parts it can make all sorts of things that are useful in and of themselves. People who develop RepRap for its own sake are also building a foundation for a tool that will empower people who have no interest in RepRap itself, but rather in what it can do for them. These users are part of the dynamic open source community; stress testing and providing heterogeneous, massively parallel feedback. This is analogous to the open source software suites that run the internet. Software like Apache enables all the activities of people who are not interested in Apache.

There is strong empirical support for the idea that open source development results in more resources being applied to a problem. RepRap developers reported between 145 and 182 equivalent full-time developers as compared to Stratasys (industry leader) having 361 full-time employees total, many of which are in marketing, customer support, and sales. A different report shows that 6.2% of United Kingdom consumers did some kind of non-job-related innovation and spent 2.3 times the total expenditure for research and development of consumer products of all firms in the United Kingdom.

It is not necessary to have a RepRap to participate in the digital, online development of physical objects, although it does make it easier to personally capture the benefits. But due in no small part to the RepRap project itself, it is likely that this capability will only become easier for the average person to obtain in the near future. This bodes well for the future of open hardware projects.

In fact, the additive manufacturing industry saw relatively stable sales during the recession, but RepRap and derivative systems saw impressive growth. This is empirical support for the idea that open source entrants to an established and mature market can fill a void that nobody knew existed.

Analysis

The RepRap project, by indicating that there are no limits to the technology that can benefit from open source, supports point 3.1 in the appendix which supports the third major point that space technology should be able to benefit from open source development. It also supports point 3.2, that open source can successfully penetrate mature markets like rapid prototyping.

Perhaps the most interesting aspect of the RepRap project is that it did not start out as an engineering project[12]. It started with the realization that the vast majority of organisms depend on other organisms, to a greater or lesser extent, for their own reproduction. Many experiments in artificial reproduction presupposed that the artificial entity will be dependent only on itself for reproduction, but this is a strategy that has long been marginalized in nature. If an artificial entity were to look to another organism for reproductive help, it makes sense for that organism to be humans. Mutual aid is a common reproductive strategy; flowers cannot reproduce without bees, which they reward with nectar. Something machines are good at is doing the same precise thing over and over again, which humans cannot do, and humans are good at tasks which require dexterity, which machines are awful at. So a strategy of mutual aid would imply a machine making a kit of its own parts and depending on a human to put them together. For their trouble, the humans would be rewarded with parts useful for other things. Open source was adopted because anything which seeks to extract payment prior to reproducing itself will be at a disadvantage compared to something that gives its own design (DNA) away for free. The designs were released under a free software license, and from there the RepRap began to exponentially increase its population.

12 Jones, R., et. al. (2011). RepRap – the replicating rapid prototyper. *Robotica*, 29, pp 177-191 doi:10.1017/S026357471000069X. Retrieved from http://journals.cambridge.org/action/displayFulltext?type=1&fid=7967176&jid=ROB&volumeId=29&issueId=01&aid=7967174&bodyId=&membershipNumber=&societyETOCSession=

This is supports point 1.4 in the appendix which supports the first major point that open source, rather than being focused on cost, is focused on solutions to technical problems.

As a piece of technology, RepRap occupies a unique niche. Rapid prototyping machines take digital files and autonomously convert them into physical objects. If the source code of a program is what a human can alter, and it gets compiled into a new form that can actually be executed by the computer, then there is a strong analogy to RepRap. The 3-dimensional Computer Aided Design (CAD) files that people work with to design parts are like the source code, and RepRap is like the compiler in that it converts the source code (CAD files) into a form that can actually be used (object). To further cement the analogy, the code produced by a compiler is even called object code. A similar analogy can be drawn with the Arduino microcontroller profiled earlier. The programs that run on Arduino are called images, and it is the Arduino that converts them into a useful form. This allows a direct comparison between the process followed by open source software developers, to open source electronics developers, and to open source machine developers. In each case there is a design file that must be “compiled” before someone can extract the benefit from it.

Software came first because it has the lowest associated cost. It is mostly just converting logic into a different form. The tools for software development started out proprietary, but open source made them free or effectively free. Those software tools were used to open source electronics development tools, which cannot be free, but can be cheap enough to be replaced for every new project, which is common among Arduino developers. The open source electronics development tools were used to open source machine development tools. Rapid prototypers allow a developer to directly convert their digital file into a machine component, sometimes even a complete mechanism, just like a compiler allows a software developer to convert their program design into object code the computer can execute. RepRap is the first step in allowing developers to personally capture the benefits of open source hardware because it negates the need to personally fabricate something. Just like programmers do not program directly in machine language, with a RepRap builders do not have to personally carve out a part.

If open source principles started with software, and were then manifested in electronics via the Arduino, and were then manifested in mechanisms via the RepRap project, perhaps the logical conclusion is the Open Source Ecology (OSE) project. OSE is a non-profit that is developing open source alternatives to all of the heavy agriculture and infrastructure machinery necessary to sustain a modern standard of living[13]. However, while open source and non-profit goals often align, there are more than a dozen examples of for-profit businesses based entirely (or in part) on open source hardware that are making more than one million dollars a year in revenue[14]. The pattern of open source development being successfully applied to ever more expensive endeavors is clear and unambiguous.

This is point 2.4 in the appendix and supports the second major point that the only relevant difference between different open source domains is the amount of resources necessary to convert designs into useful objects. Converting software costs less than making circuit boards which costs less than fabricating machines, but the development principles are identical.

Transition

The RepRap project encapsulates the analogy between the remarkably successful open source software movement and the potential for applying the same principles to hardware. The next article examines space technology that could benefit from the open source approach.

13 http://opensourceecology.org/about.php

14 http://singularityhub.com/2010/05/10/13-open-source-hardware-companies-making-1-million-or-more-video/

Esper, J. (2005). Modular, Adaptive, Reconfigurable Systems: Technology for Sustainable, Reliable, Effective, and Affordable Space Exploration. *Space Technology and Applications International Forum*, 1033-1043.

Summary

It is stating the obvious to point out that spacecraft are expensive, and anything we can do to reduce cost will be beneficial. Failing that, reducing system complexity, integration and test times, and increasing flexibility will also help and will probably lead to cost savings anyway. The Modular, Adaptive, Reconfigurable System (MARS) technologies are supposed to do just that. The National Aeronautics and Space Administration (NASA) tried modularity before. In the 1970s it was called Multi-Mission Modular Spacecraft (MMS). It was able to reduce integration and test times by 50-80% but it lacked a process for upgrading the technology over time so it gradually faded into obsolescence.

Technology has improved over the last several decades, which allows for new functions that were impossible for MMS to achieve, but it is the incorporation of an on-going process for evolving MARS that will make it truly successful. The key to implementing the process is modularity, which allows cost savings through the maximization of known, standardized components. The primary area of standardization will be at the interface, which permits module exchange and plug-and-play capabilities, something not yet implemented in space systems (g. 1037). It is vital that a critical mass of spacecraft developers use the standard or else the “someone else's design” mentality will lead them all down incompatible paths.

In MARS, “adaptive” means that components can learn about the context they find themselves in and change to meet it, “reconfigurable” means the entire system must be able to morph to fit different missions, and “system” means that it should be flexible enough for each individual developer to define whatever system-level is relevant to them at the time. Flexibility in modularity level (chip or circuit board or box, etc) will be a key strength. As much as possible, MARS should build off of existing commercial standards. Standards like Ethernet, Firewire and USB already exist; mechanical and fluid interfaces will have to be developed or borrowed. Existing open source operating systems like Linux should be used so as to encourage industry involvement.

NASA has always pioneered new technology and there is a void left by MMS. That void can be filled by MARS if leadership emerges to guide a broad and integrated approach.

Analysis

This supports point 3.3 in the appendix in that it is a technical problem, which open source is particularly good at solving, which supports the third major point that space should be able to benefit from open source development.

The Defense Advanced Research Projects Agency (DARPA), which is tasked with looking for into the future so as to obtain the benefits of the technological cutting edge as quickly as possible, has come down firmly on the side of open development in at least one project. The Adaptive Vehicle Make (AVM) project is an attempt to reduce by a factor of 5 the amount of time it takes to produce complex defense systems like fighting vehicles. DARPA decided that the only plausible way to accomplish this feat is with open development. The software necessary to simulate cyber-electro-mechanical systems, collaboratively engineer them over the internet, and confirm that they will work before they are ever constructed, will be substantial and valuable. Despite the inherent value of that suite of tools, DARPA required the contractors to deliver unlimited rights to the government because the whole thing would be released under an open source license[15]. Additionally, and of particular relevance to this paper, DARPA specifically listed "space systems" as one of the types of vehicles that could be designed by a crowd on open source software.

An excellent example of the power of open standards is the International Docking System Standard (IDSS)[16]. It was produced by the member-entities of the International Space Station

15 DARPA-BAA-11-21. (2010). *Broad Agency Announcement vehicleforge.mil*. Retrieved from https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&ved=0CDkQFjAA&url=https%3A%2F%2Fwww.fbo.gov%2Futils%2Fview%3Fid%3De68d627434c705fe8e3abf0b9b6b74b3&ei=PwbIUKmaBs3DyQH06ICQBQ&usg=AFQjCNEkJ-lVQNO3MChz1FziZqMSSbVhTw&bvm=bv.1354675689,d.aWc

16 IDSS IDD. (2011). *International Docking System Standard (IDSS) Interface Definition Document (IDD) Revision A*. NASA, Roscosmos, ESA, CSA, MEXT. Retrieved from http://www.internationaldockingstandard.com/download/IDSS_IDD_RevA_Final_051311.pd

(ISS) to allow for crew rescue and general endeavors by different spacecraft. The standard defines such things as the physical geometry of the interface and the magnitudes of different loads it can handle. What is notable is that the standard provides “only the requirements necessary” and so “allows designers the flexibility to design and build docking mechanisms to their unique program needs.” (p. 1). In fact, the explicit purpose of the IDSS IDD is “to provide basic common design parameters to allow developers to independently design compatible docking systems.” (pg. 1).

While the IDSS IDD does not mention a copyright, let alone an open source license, the principles of community and modularity are front and center. This is point 2.1 in the appendix which supports the second major point that open source development of hardware follows the same principles as software.

Transition

Modularity and community-wide standards are core competencies of open source, and they are sorely lacking in space technology. The next article illuminates the theory behind low-cost space systems and calls for new policies that open source could fulfill.

f

Wertz, J. R., & Larson, W. J. (1999). Design of Low-Cost Spacecraft. In J. Wertz, & W. Larson (Eds.), *Space Mission Analysis and Design Third Edition* (pp. 853-881). Location: Space Technology Library

Summary

Low-cost spacecraft are characterized by smaller and simpler design, a result of reducing the mission scope. With less cost involved, and a smaller mission, the user can take more risks with new technology which, in practice, tends to actually increase system reliability. However, in a broad sense, reliability is increased through simplification, rather than via redundancy like on expensive systems. Smaller spacecraft also reduce launch costs, which can be half of the cost of a satellite system.

There are numerous examples of successful cheap satellites. The very first satellites, like Sputnik, used extremely limited and cheap hardware, although if the cost of developing the entire program is factored in they do not look so cheap. The OSCAR-series satellites, constructed within severe budget constraints by teams of volunteers, have been extremely successful. Additionally, many nations which cannot afford a full space program have launched and operated cheap spacecraft.

Ultimately, the most important factor in the final cost is the performance that is required of the system; you get what you pay for. That being said, it is possible to maximize your dollars spent on spacecraft by using more on-board processing, requiring less pointing accuracy, and reducing system power. Other ways to minimize cost is to reduce required launch site support, allow for integration in to multiple launch vehicles, and provide for maximum on-orbit flexibility. A strategic approach of reducing system complexity also leads to lower cost through fewer failures because modern technology fails far less than humans trying to integrate and utilize that technology in a complicated way. A smaller project also benefits from a smaller, less expensive team, the camaraderie of which can actually induce more and more creative work, further lowering costs. Keeping things small, and designing conservatively, also creates cost benefits when the thermal, mechanical and other characteristics of the system do not require extensive and specialized testing, analysis and integration.

Wertz wraps up the last chapter in Space Mission Analysis and Design (3rd ed) with a call to action. He asserts that the modern limits on space exploration are not technological or financial, they are just policy. He proposes that the challenge of the early 21st century is “becoming more efficient and cost effective in our use of space.”

Analysis

If, as Wertz asserts, the primary barrier to space exploration is the lack of a policy which leads to greater efficiency, then open source is a viable candidate for space technology development. The primary drawback to open source space is that developers are motivated by capturing the benefit of their work, and it is a bit of a stretch to imagine individuals personally capturing the benefit of space technology. It is not, however, unprecedented.

Starting from the ground up, so to speak, is Copenhagen Suborbitals[17]. CS is a non-profit and open source organization that runs entirely on private donations. They are based in Denmark, pretty much all of the work is done by the two founders, and they even have their own sea-launch platform. Allegedly, CS has achieved the record for the most powerful amateur rocket ever flown and for the first amateur rocket with a full-sized crash-test dummy passenger[18]. Their goal is to launch passengers on suborbital flights and, eventually, to place payloads into Low

17 http://www.copenhagensuborbitals.com/mission.php

18 http://translate.google.com/translate?hl=en&rurl=translate.google.com&sl=da&tl=en&u=http://www.raketvenner.dk/

Earth Orbit (LEO). Like all open source projects, CS tends to be behind in their documentation, and like all space projects they are limited by the export control laws of their country, but one example of the sort of work CS is open sourcing is Project GALCIT[19]. Their calculations indicate that if their HEAT 2X booster had assistance from solid propellant upper stages it would be able to place micro satellites in LEO (p. 4). However, there are difficulties associated with solid rockets. The most significant difficulty is that the ingredients are expensive and rare and, worst of all, the resulting rocket is extremely delicate. Even a single crack in the structure of the fuel could create a catastrophic failure, so solid rockets cannot be made any bigger than they an be safely transported. By digging up the chemistry of old Jet Assisted Take Off (JATO) rockets, and conducting a few experiments, CS documented and published a process for producing the fuel and motor of a crack-resistant solid-propellant rocket. Extrapolating from experimental data indicates that scaling up CS's JATO-inspired solid rocket motor would allow them to orbit a 200 gram satellite using their existing launch vehicle (p. 19).

A payload that might ride on CS's open source launch vehicle, when it gets a little more powerful, is the Arduino-based open source satellite ArduSat[20]. This is a project that recently received 3-times its requested funding on Kickstarter, a crowd-funding website. 676 individuals pledged between $1 (130) and $10,000 (1) to help build and launch the modular, open CubeSat. Users can create their own software which is then uploaded to the orbiting satellite which accomplishes whatever mission the user wanted. Some of the things ArduSat will be capable of are detecting meteors, generating truly random numbers, earth observation, and radiation detection

19 Madsen, P. (2012). *Project GALCIT, Experimental evaluation of a simple solid composite propellant*. Copenhagen Suborbitals. Retrieved from http://www.copenhagensuborbitals.com/public/GALCIT_Report.pdf

20 http://www.kickstarter.com/projects/575960623/ardusat-your-arduino-experiment-in-space

games[21].

Copenhagen Suborbitals, which will launch tourists into near-space or payloads into LEO, and ArduSat, which will allow individuals to conduct experiments in orbit, are examples of how it is possible for individual (or small groups) to personally capture the benefits of space technology. There are also many more examples of open source space organizations and projects. "In the last four years nearly one dozen groups have formed with the stated purpose of developing space flight systems in a manner similar to that of open source software projects.[22]" This is point 3.1 in the appendix and supports the third major point that space technology should be able to benefit from open source.

21 https://docs.google.com/file/d/0B4hRbWIH9kinS3FfOElJTU5Gc00/edit

22 Simmons, J., Black, J., Moran, G. (2011). *A Survey of the Open Source Spaceflight Movement*. AIAA Space 2011 Conference & Exposition. DOI: 10.2514/6.2011-7225 Retrieved from http://arc.aiaa.org/doi/abs/10.2514/6.2011-7225

Conclusion

This paper lays out the results of my research attempting to answer the hypothesis: If the use of open source licenses in one area of technology, like rapid prototyping, can produce significant cost savings, then a more open approach can produce similar cost savings in another area, like space.

My research was filtered through five class subjects from my Master's program: SPSM 5600 – Space Systems Acquisition Law, SPSM – 5650 Space Systems Contracting, SPSM – 5750 Space Systems Engineering, SPSM 5770 – Space Operations Management, and SPSM 5900 – Space Commercialization.

Unfortunately, I do not feel I was able to directly answer the hypothesis. What I was able to do was support three major points which, taken together, make a strong analogous argument for an affirmative answer to the hypothesis.

First, the original proprietary paradigm of for-profit commercial entities created the legal structure of copyright so as to enforce the top priority of maximizing cost to the users. When software source code began to be kept secret under this legal standard it inspired a cultural backlash. The free paradigm rose up with exactly the opposite top priority, which was to minimize cost to the users. These two cultures found it impossible to cooperate since they prioritized diametrically opposed standards. The open paradigm emerged as a compromise between the two. The mutually acceptable solution was to prioritize solutions to technical problems first and let the cost issue work itself out. Thus, open source preserves most of the ideals of the free software movement, and maintains the capability to coordinate the efforts of intrinsically motivated volunteer developers, while being pragmatic enough to be adoptable by for-profit businesses.

Second, the principles of open software, which are well proven, and open hardware, which is still taking its first steps, are identical. Both processes share modularity and community as core competencies. Nothing suggests that there is a significant difference between the government's, or any entity's, rights in software and hardware. Although it is theoretically possible for patent law to interfere due to its fundamentally different nature compared to copyright, in practice copyright will always apply to designs, but patents are few and far between. Ultimately, the only difference between software, electronics and machines (hardware) is cost. It is effectively free to convert software from a design into a useable object, it costs more to do it in electronics, and it costs even more to do it for a machine. This dynamic is supported by the observed progression during which open source software formed the foundation for open source electronics which formed the foundation for open source machines.

Third, there is no indication that the benefits of open source are limited to a particular type of technology or a particular domain. The greatest benefit does seem to come from tools, standards and infrastructure, although that is probably following from the fact that development tools are used more often than the things that are developed with them. Basically, you see the greatest benefit in improvements to the things you use the most often. These benefits have been observed penetrating mature proprietary industries, which indicates the barriers to entry into space exploitation via open source development can be overcome. As a target for the growing open source movement, space is uniquely appropriate. It is an environment that rewards technical solutions over all other considerations and it has traditionally been reserved as a common frontier that humanity can and should cooperate in exploiting. Even as this paper is being written open source pioneers are capturing the benefits of space for themselves and others.

Since I was unable to find sufficient empirical evidence to produce a rigorous answer to the hypothesis, I suggest the following questions be answered with further research. First, what is the relationship between the cost of open source hardware projects and their proprietary equivalents? Second, what is the legal status of hardware-specific open source licenses? Third, Is there a difference between the motivations of open source software and hardware developers?

Appendix

1. Open source is a development paradigm that has as its top priority the optimal solution of technical problems.
    1. This can be contrasted against the proprietary paradigm (traditional for-profit business) which prioritizes maximizing cost to the users so as to achieve the highest possible profit.
    2. This can also be contrasted against the free paradigm (free software) which was, and still is, intentionally in direct opposition to the proprietary paradigm in that it prioritizes minimizing cost to the users for moral reasons.
    3. All three of these paradigms start with a top priority, develop a community around it, and use the law to enforce their ideal. The reactionary free paradigm, and the pragmatic open paradigm, actually use the legal structure originally intended to enforce proprietary ideals in reverse to undermine or compromise those ideals.
    4. Open source developers are most strongly motivated by the intrinsic desire to overcome a challenge. The strongest extrinsic motivation is status within the community; basically recognition by the community that they have overcome a lot of challenges.
2. The only substantial difference between open source anything is in the amount of resources necessary to transition between design and implementation.
    1. The principles of modularity and community are identical.
    2. Nothing suggests that there is a significant difference between the government's, or any entity's, rights in software and hardware.
    3. In theory, actual objects could introduce patent law concerns in addition to the

    copyright that covers their designs. In practice, this is unlikely to manifest, indicating that the major concern of open hardware will be the copyright law that covers the designs, just like software

    4. The cost difference between software design and implementation is small; usually just a matter of translating logic into a different form. The design and implementation of electronics has a greater cost difference; usually requiring some amount of professional fabrication. Actual machines have a significant difference between their design and their implementation costs. This is well represented in the timeline of open source software, electrical and mechanical systems.

3. Space technology should be able to realize substantial benefits from adopting the open source development paradigm.
    1. There is no indication that the benefits are limited by the type of technology, although the greatest benefit does seem to come from application to shared tools, standards and infrastructure.
    2. Open source of all varieties has demonstrated an ability to penetrate large, mature markets. Often this manifests as cheaper/easier access to capabilities that already exist.
    3. Space, being a particularly unforgiving environment, and being traditionally exploited as a frontier for all humanity, is perfectly suited to embrace the open source development paradigm's focus on prioritizing the best technical solution over all other considerations.
4. The hypothesis was: *If the use of open source licenses in one area of technology, like rapid prototyping, can produce significant cost savings, then a more open approach can*

produce similar cost savings in another area, like space.

1. The three major points, open source focuses on optimal solutions to technical problems, no fundamental difference between the development principles of software and hardware, and the applicability of those principles to the space domain, seems to support the hypothesis. There is little reason to think that open source savings in software cannot be realized in hardware. There is also little reason to think that the same or similar savings cannot be realized in any technological area.

5. I suggest that peer-reviewed research of the following questions would be valuable.
    1. What is the relationship between the cost of open source hardware projects and their proprietary equivalents?
    2. What is the legal status of open source hardware licenses?
    3. Is there a difference between the motivations of open source software and hardware developers?