Perspectives on Code Forking and Sustainability in Open Source Software

Linus Nyman, Tommi Mikkonen, Juho Lindman, Martin Fougère

Perspectives on Code Forking and Sustainability in Open Source Software

Linus Nyman[1], Tommi Mikkonen[2], Juho Lindman[1], and Martin Fougère[1]
1 Hanken School of Economics, Helsinki, Finland
firstname.lastname@hanken.fi
2 Tampere University of Technology, Tampere, Finland
tommi.mikkonen@tut.fi

Abstract. The ability to create high-quality software artifacts that are usable over time is one of the essential requirements of the software business. In such a setting, open source software offers excellent opportunities for sustainability. In particular, safeguarding mechanisms against planned obsolescence by any single actor are built into the definition of open source. The most powerful of these mechanisms is the ability to fork the project. In this paper we argue that the possibility to fork serves as the invisible hand of sustainability that ensures that code remains open and that the code that best serves the community lives on. Furthermore, the mere option to fork provides a mechanism for safeguarding against despotic decisions by the project lead, who is thus guided in their actions to consider the best interest of the community.

1 Introduction

Sustainability is a concept which is often automatically associated with open source software. Access to the source code allows developers to build solutions that are better protected from potentially harmful actions of any single developer, company, or organization. The openness of the source code also means that decisions concerning the software artefact become transparent to the developer community.

In this paper we address the role of code forking – a situation in which several versions of a piece of software originating from a single, shared code base are developed separately – in ensuring the long-term sustainability of a software system. Furthermore, we advocate the freedom that enables developers to create novel features that may go well beyond what the original developers anticipated. This freedom, a key factor in the promise of open innovation that builds on open source software, can nurture open source projects through difficult times and extreme events that could otherwise prove lethal. An example is a hostile acquisition, which may cause radical changes in the project.

2 Sustainability and Planned Obsolescence

The link between software and sustainability is not evident if considering sustainability as something related to raw materials or energy in design, use or maintenance [1]. Indeed, sustainability is an "essentially contested" concept [2, 3, 4], and thus sustainability of a product can be interpreted in many ways. We take the view of the consumer and focus on two central elements: quality and staying power – how to create a high-quality product that is usable as long as possible.

This approach to product sustainability conflicts with what is known as "planned obsolescence", a term popularized in the 1950s by American industrial designer Brooks Stevens [5]. Stevens defined planned obsolescence as the act of instilling in the buyer "the desire to own something a little newer, a little better, a little sooner than is necessary" [6]. From the fashion industry, where last year's models are designed to look out-of-date by the time this year's models come around, to the software industry, where the norm is for software to be compatible with older models but not with newer ones, planned obsolescence – considered by some "an engine of technological progress" [7] – has become an inescapable part of the consumer's everyday life, which is increasingly problematized in business ethics literature [8].

Digital artifacts, of course, differ substantially from the end products of 1950s industrial design, or even those of today. The main differences are related to their characteristics as editable, interactive, reprogrammable, distributed, and open [9]. These characteristics dictate that software as an artefact is prone to being changed, repaired and updated rather than remain fixed from early stages of the design process (see also [10]). The software marketplace has transferred planned obsolescence to the digital realm by creating ways to benefit from these artefact characteristics. The revenue models of companies that operate in the software marketplace thus welcome versioning, lock-ins, competition, and network effects [11].

Open source software offers an alternative to some of the pitfalls of planned obsolescence. Rather than needing to buy something "a little newer, a little better", the open source community can simply make the existing product a little – or a lot – newer and better. In open source, anything, once invented, once written, need never be rewritten. On the other hand, the software product is never ready but can become stable and mature enough for the developer community. With community interest, the software can always be improved.

The right to improve a program, the right to make it portable to newer as well as older programs and versions, and the right to combine many programs into an even better entity are all fundamental privileges built into the very definition of open source, and these rights are often exercised by the involved parties [12]. Therefore, in open source systems any program that has the support of the open source community will enjoy assured relevance rather than planned obsolescence. In fact, planned obsolescence in open source is impossible without community consent, due to a practice which is at once both the sustainer and potential destroyer of open source programs: the code fork.

3 Code Forking

A popular metaphor in economics is Adam Smith's "invisible hand", a self-regulating force that guides the marketplace [13]. We claim that open source software has its own invisible hand: the fork. In fact, even the *possibility* of a fork usually suffices. A broad definition of a code fork is when the code from an existing program serves as the basis for a new version of the program; more specifically, a version which seeks to continue to exist apart from the original[1]. Forking can (though need not) be the result of a split in the developer community regarding the software artefact, its development practice, or the direction of the development, and is in such cases usually followed by a split in the user community. With open source, one can always fork a project: an inclusion of the right to fork is a prerequisite of all open source licenses. Furthermore, the licensing terms impose no conditions which would in any way require developers to adhere to the original development line.

Forking is paradoxical in nature; it is simultaneously both one of the greatest threats that an individual project faces, and the ultimate sustainer: a guarantee that as long as users find a program useful, the program will continue to exist. The threat to the program comes mainly in the form of the (potential) dilution of both users and developers. As Fogel [15] has noted, it is not the existence of a fork that hurts a project, but rather the loss of developers and users. The benefits of a fork come in ensuring that the program can continue to exist regardless of external circumstances. If, for instance, the developers of a program under a permissive license decide to relicense it under either a proprietary license or a license otherwise perceived to be less favorable, the community can fork a new version and continue development. Forks can also serve as an escape hatch for projects and developers who find themselves cornered or unable to continue on a planned course. In the case of a program remaining under an open source license, but where the people or company shepherding the code make decisions which run counter to the interests of the larger community and developers, forking ensures the continued development, as the community and developers can fork a new version on which to continue working[2].

While there are no guarantees that a fork will become accepted or used by the community – forks of popular software in particular are likely to face considerable obstacles to their sustainability in the form of trademarks and the brand value of the main branch – the mere possibility of forking a program has a huge impact on how open source programs projects are governed [15]. A better-managed project increases chances of sustainability – even a project viewed as important and necessary can become unsustainable if people no longer want to be a part of the group working on the program. In successful projects, however, a dynamic seems to exist where

[1] A branch is problematic to categorize at the time of its 'creation': it can be considered a fork if it is not, at some later point, merged back into the main branch. The intricacies of comparing and defining forking versus fragmentation, light forking, 'pseudo-forking' [14], branches, and versions is the topic for a paper under development, but beyond the scope of this one.

[2] In recent years, examples of using a fork for the sustainability of a community include high-profile cases such as the forking of OpenOffice into LibreOffice and the creation of various projects from the code base of MySQL.

developers are happy enough to follow the project leader as long as the project leader listens to developers' views enough to keep them on board: while the individual members of the development team all *could* fork the program, they choose not to. This balance creates continuity for long-term cooperation.

4 Code Forking and Sustainability

The first of Lehman's laws of program evolution is that of change or decay – a program must continue to evolve in order to remain useful [16]. Brooks notes not only that all successful software gets changed, but also that successful software "survives beyond the normal life of the machine vehicle for which it was first written" [10, p. 185]. Forking can offer solutions to the aforementioned concerns. The possibility of forking provides the community with the tools it needs to handle situations in which a program could become obsolete (for the community as a whole or a particular segment of the community) either through stagnation, a change in licensing, or any other reason by enabling the creation of a new version of the system, a porting to a new hardware environment, a change in program focus, and many other possible solutions to avoid decay and obsolescence (see [17] for examples).

If several developers leave a project and start their own fork, benefits to sustainability can still be found. Among the more obvious is that the developers are still working on the program, be it on a different version. Had forking not been possible, they might have stopped their work on the program entirely. Also, as long as the licenses are compatible, any breakthroughs or developments in a fork can be incorporated into the original version. While there may be duplicated efforts involved, all versions can still benefit from the work done on others.

Given that the reuse of existing code is a common practice in open source [18], one could contrast the evolution of code with the evolution of species since open source software, like living species, can be seen to "reproduce" and pass on certain traits through forking and reuse. In discussing natural selection, one of the central tenets resulting from Charles Darwin's research, Darwin notes that each variation of a species, if useful, is preserved, while "any variation in the least degree injurious would be rigidly destroyed" [19, pp. 130-131]. The same can be said of code forks – if a new variation is considered useful by developers and community it will endure, while forks considered "injurious", or at least less favourable than an available alternative, will not endure[3]. Open source, however, is not as unforgiving as Darwin's nature in the sense that even if a program falls into disuse, it may still continue to exist (for instance in the form of source code on a forge). An abundance of similar yet unique forks of the same program may prove useful for its

[3] Variations which are "neither useful nor injurious […] would not be affected by natural selection, and would be left a fluctuating element", Darwin [19, p. 131] notes. In the case of code forking, these "fluctuating elements" could conceivably become either useful or injurious in the event of new developments in the environment.

sustainability through an increased likelihood of survival if some forks, by chance or design, are better protected than others against adversity, be it in the form of a virus, unfavourable corporate or community actions, or any other form.

A greater amount of similar yet distinct forks may also help bring about new functionalities, even innovations. Disruptive innovations – innovations which improve a product or service in a way that the market does not expect, eventually displacing the earlier technology – are commonly not so much advances in technology as they are new combinations of existing technology [20]. Code forking as a practice could both create programs better suited to benefit from disruptive innovations by other actors, as well as create enough building blocks – variations of programs – to make new functionalities as well as innovations more likely to occur. Indeed, is there any other area or field in which the combining in new ways of existing technologies, in this case computer programs, is as catered to and as ingrained in community practice as in open source development? The plethora of forges online offer hundreds of thousands of programs, available for forking and reuse in any new, creative way the user can imagine.

Perhaps the greatest potential threat to the practice of forking and combining different open source programs is the question of license compatibility. The so-called copyleft, or viral, licenses, chief among them the GPL family of licenses, set restrictions regarding which types of licenses they can be combined with, while permissive (or non-viral licenses) like the BSD and the MIT licenses impose no such restrictions (see, for instance, [21, 22, 23]). For practical use, there are well-established ways to overcome some of the restrictions [24].

For an open source project to remain sustainable it must evolve with its user base. The same goes for the developers, whose actions must also evolve along with the evolution of the project as well as the users. The possibility to fork is one of the key factors that ensure that open source will continue to evolve and thus remain sustainable. Open source programs can also cease to develop; some programs and pieces of code live on while others die out. Forking, as well as the effect of the possibility of forking, ensures that the selection lies in the hands of the community. At its best, open source software, guided by the invisible hand of forking, may well render planned obsolescence itself obsolete.

5 Conclusions

Forking has the capability of serving as an invisible hand of sustainability that helps open source projects to survive extreme events such as commercial acquisitions, as well as ensures that users and developers have the necessary tools to enable change rather than decay. Code forking may also have other, less foreseeable benefits, as some variations of a program may be better suited either to surviving adverse events or to aiding in achieving new functionalities and innovations, for instance through the novel combining of programs. The possibility of forking is a powerful incentive for ensuring continuity and the long-term viability of an open source development, and thus the sustainability of the resulting software artefacts.

References

[1] Murugesan (2008) Harnessing Green IT: Principles and Practices. IT Professional, vol. 10, no. 1, pp. 24-33, Jan./Feb.

[2] Connelly (2007). Mapping Sustainable Development as a Contested Concept. Local Environment: The International Journal of Justice and Sustainability, Vol. 12, No. 3, pp. 259-278.

[3] Davison (2001) Technology and the contested meanings of sustainability. State University of New York Press, Albany, NY.

[4] McManus (1996) Contested terrains: Politics, stories and discourses of sustainability. Environmental Politics, Vol. 5, No. 1, pp. 48-73.

[5] Planned obsolescence, The Economist, 23 March 2009. Available at: http://www.economist.com/node/13354332, accessed 14 September 2011.

[6] Brooks Stevens biography, available at: http://www.brooksstevenshistory.com/brooks_bio.pdf, accessed 14 September 2011

[7] Fishman, Gandal and Shy (1993) Planned Obsolescence as an Engine of Technological Progress. Journal of Industrial Economics, Vol. 41, No. 4, pp. 361-370.

[8] Guiltinan (2009) Creative Destruction and Destructive Creations: Environmental Ethics and Planned Obsolescence. Journal of Business Ethics, Vol. 89, pp.19–28.

[9] Kallinikos, Aaltonen and Attila (2010) A theory of digital objects. First Monday, Volume 15, Number 6-7, June.

[10] Brooks (1995) The mythical man-month. Addison-Wesley, Boston, MA.

[11] Shapiro and Varian (1998) Information Rules: A Strategic Guide to the Network Economy. Harvard Business School Press, Boston, MA.

[12] Fitzgerald (2006) The Transformation of Open Source Software. MIS Quarterly, vol. 30, no. 3, 2006, pp. 587–598.

[13] Smith (1776) The Wealth of Nations (Bantam Classic Edition March/2003). Bantam Dell, New York, NY.

[14] Raymond (2001) The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary. O'Reilly, Sebastopol, CA.

[15] Fogel (2006) Producing Open Source Software. O'Reilly, Sebastopol, CA.

[16] Lehman (1980) Programs, Life Cycles, and Laws of Software Evolution. *Proc. IEEE* 68 (9): 1060–1076. available at: http://www.cs.uwaterloo.ca/~a78khan/cs446/additional-material/scribe/27-refactoring/Lehman-LawsOfSoftwareEvolution.pdf.

[17] Nyman and Mikkonen (2011) To Fork or Not to Fork: Fork: Fork Motivations in SourceForge Projects. Proceedings of the 7th International Conference on Open Source Systems (OSS 2011), 259-268, Springer.

[18] Haefliger, von Krogh, and Spaeth (2008). Code Reuse in Open Source Software. Management Science, Vol. 54, No. 1, pp. 180-193.

[19] Darwin (1859) The Origin of Species (1985 Penguin Classics edition). Penguin Books, London, England.

[20] Christensen (1997) The Innovator's Dilemma. Collins, New York, N.Y.

[21] Meeker (2008) The Open Source Alternative: Understanding Risks and Leveraging Opportunities. Wiley, Hoboken, NY.

[22] Sinclair (2010) License Profile: BSD. International Free and Open Source Software Law Review, vol. 2, Issue 1. DOI: 10.5033/ifosslr.v2i1.28.

[23] Lindman, Rossi and Puustelli (2011) Matching Open Source Software Licenses with Corresponding Business Models. IEEE Software, vol. 28, no. 4, pp. 31-35, July/Aug

[24] Hammouda, Mikkonen, Oksanen and Jaaksi (2010) Open Source Legality Patterns: Architectural Design Decisions Motivated by Legal Concerns. Published in the proceedings of the 14th International Academic MindTrek Conference: Envisioning Future Media Environments. ACM, New York, NY.