Does Firefox obey Lehman's Laws of software Evolution?

Yan Dong

Masters Candidate

Department of Management Sciences

University of Waterloo

Waterloo, ON, Canada

y4dong@engmail.uwaterloo.ca

Shahab Mohsen

Masters Candidate

Department of Computer Science

University of Waterloo

Waterloo, ON, Canada

smohsen@cs.uwaterloo.ca

Abstract

Maintenance, bug fixes and software updates are necessary to every software project in order to keep it competitive in the software market. Lehman [2] called this process the evolution of a software system and has found eight patterns describing the evolution process for software systems. Michael Godfrey and Qiang Tu [7] provided a counter example against the patterns Lehman provides. They claimed that his laws do not necessarily apply to open source projects and provided Linux operating system kernel as an example.

In this paper we tried to extend the study of Godfrey and Tu to Firefox , another well-known open source project, and try to find out if Lehman's laws hold for Firefox or not. We also try to provide an improvement to Lehman's laws which tries to fix the existing problems.

1. Introduction

Evolution is inevitable for software systems. Due to various reasons, software systems need to change. Lehman [2] believed that software is a tool to mechanise the real world. He claimed that since the real world is dynamic so the software should get changed and updated to be able to reflect the changes in the real world. Therefore, bug fixes, updates and minor changes are inseparable parts of the life cycle of software. But there still seems to be some existing problems. In many cases, programmers fail to provide good documentation for the code they write [14]. As a result, it will become harder for the maintainers to change the code as the code increases in size. Lehman [2] also recognized some other facts as patterns which is true for every software project he has seen. After some experimentation, he gathered his experience through his more than 30 years of professional work and observation in software engineering field and introduced his eight laws of software evolution. But his methodology seemed to have some leaks. We will first introduce the problems other researchers and us have found with his methodology and results, and then we will examine his laws on a famous open source software project-Firefox. The main reason we have chosen an open source project is that, the other researchers have shown flaws in Lehman's laws when it comes to open source projects. After examining Lehman's laws we provide a brief explanation of why Firefox did or did not obey certain laws.

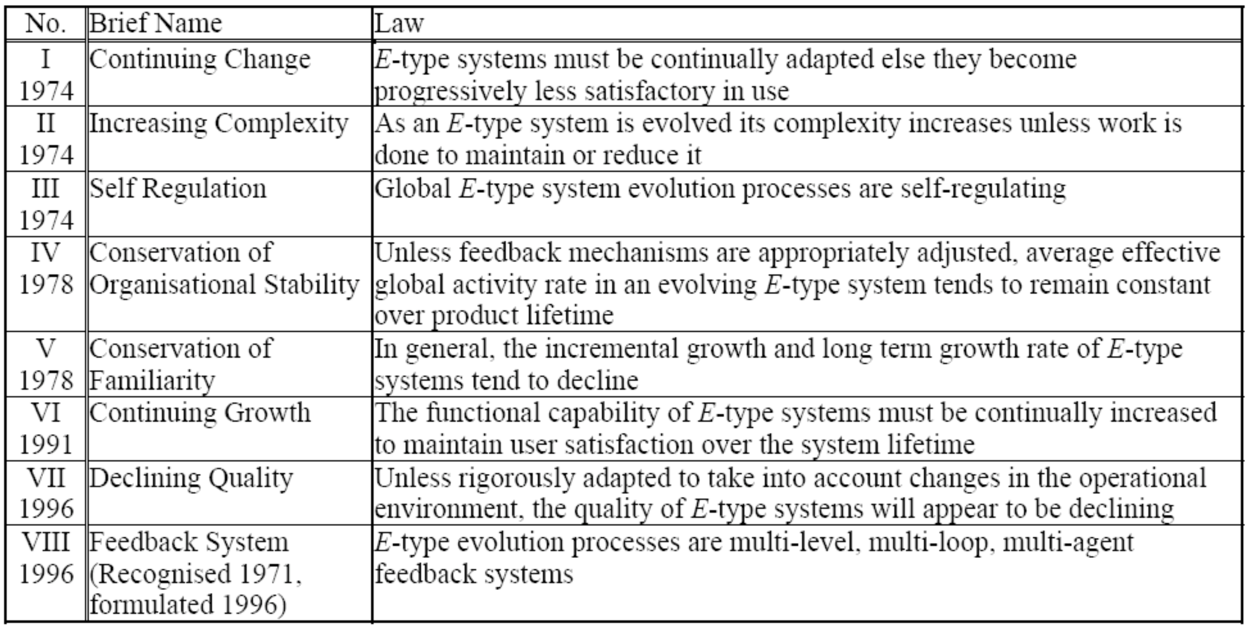
The rest of the paper is organized as follows: Section 2 presents a brief background about Lehman's laws of software evolution and the problems with Lehman's methodology and some counterexamples provided by other researchers. We will also briefly introduce the history and architecture of Firefox. Section 3 provides case studies and different results we have driven by doing experiments on Firefox. In section 4, we will compare our results to Lehman's laws and try to provide an improved set of patterns which we believe could fix the existing problems. Section 5 discusses possible topics in the future and some suggestions for further experiments.

2. Background Material

2.1 Lehman's Laws of Software Evolution

Through more than 30 years of experience and observation, Lehman found some patterns in the evolution process of different software. Lehman and Belady's [9, 10] early work on software growth dynamics and evolution concluded that evolution is an inseparable part of *large* programs. By saying large program, they meant the ones that contain between 50 and 500 thousand lines of code. The next step he chose after recognition of evolution in software systems was to categorize software. Lehman introduced three categories for existing software. First is what he called the S-type. A program is defined as being of type *S* if it can be shown that it satisfies the necessary and sufficient condition that it is *correct* in the full mathematical sense relative to a pre-stated formal *specification* (Lehman 1980, 1982). The next group he introduced is the E-Type. Type *E* programs were originally defined as "programs that mechanise a human or societal activity" (Lehman 1980). The definition was gradually improved to contain all programs that "operate in or address a problem or activity of the real world". As Lehman believes, since the real world is dynamic, and E-type software corresponds to applications in the real world, E-type software should also have the property of being dynamic which means they should evolve, change and get updated. On different kinds of E-type software, Lehman stated that "Operating systems, databases, transaction systems, control systems are all instances of the type, even though they may include elements that are of type *S* in isolation.". The last class of software Lehman introduces is Type –P: "The type was conceived as addressing problems that appear to be fully specifiable but where the users' *concern* is with the correctness of the results of execution in the domains where they are to be *used* rather than being relative to a specification."[2]

While discussing large software systems, Lehman tried to focus on Type-E software. As mentioned, He found some patterns in the evolution process of different software. He later summarized these patterns into 8 important ones and called them Lehman's Laws for software evolution. We believe it is important to notice that these laws are not the same as what we mean by the word "law". They are mainly patterns that Lehman found out that they do hold for the cases he had faced and studied on. A brief introduction and explanation has been given in figure 1 [1].



No.	Brief Name	Law
I 1974	Continuing Change	*E*-type systems must be continually adapted else they become progressively less satisfactory in use
II 1974	Increasing Complexity	As an *E*-type system is evolved its complexity increases unless work is done to maintain or reduce it
III 1974	Self Regulation	Global *E*-type system evolution processes are self-regulating
IV 1978	Conservation of Organisational Stability	Unless feedback mechanisms are appropriately adjusted, average effective global activity rate in an evolving *E*-type system tends to remain constant over product lifetime
V 1978	Conservation of Familiarity	In general, the incremental growth and long term growth rate of *E*-type systems tend to decline
VI 1991	Continuing Growth	The functional capability of *E*-type systems must be continually increased to maintain user satisfaction over the system lifetime
VII 1996	Declining Quality	Unless rigorously adapted to take into account changes in the operational environment, the quality of *E*-type systems will appear to be declining
VIII 1996	Feedback System (Recognised 1971, formulated 1996)	*E*-type evolution processes are multi-level, multi-loop, multi-agent feedback systems

Figure 1-Lehman's Laws of Software Evolution

2.2 Problems with Lehman's Laws of Evolution:

Lehman made some assumptions in order to proceed. Many of these assumptions make sense if we are studying software produced by a traditional proprietary organization. For example he stated that "The decision whether, when and how to upgrade a system will be taken by the organisation owning a product though often forced on them by others, clientele, for example. Their considerations will involve many factors: business, economic, technical and even social." It is obvious that he was not considering open source software systems, which means that open source software might not obey Lehman's Law's of evolution. Although it is true for the inhouse open source software systems that its owner could decide what to implement and upgrade, but still, open source software systems face much less business , technical and economic restrictions.

Furthermore, it is, worthy of noticing that the eight laws of Software Evolution, as outlined briefly below, are a direct outcome of such empirical observation and interpretation over a period of some 30 years."[2] Though he has also studied and examined the empirical data on the growth of the IBM OS/360-370 operating system [3, 4, 5], this might not be enough to prove the patterns he has found as "LAWS". Therefore, there seems to be some flaws in the proposal of the eight laws. To test the validity of these laws, Michael W. Godfrey and Qiang Tu [7] tried to examine these so-called laws on one of the most well know open source software systems, the Linux operating system kernel. Based on the case study they stated that "Lehman's laws of software evolution [11], which are based on his case studies of several large software systems, suggest that as systems grow in size, it becomes increasingly difficult to add new code unless explicit steps are taken to reorganize the overall design. Turski's statistical analysis of these case studies suggests that system growth (measured in terms of numbers of source modules and number of modules changed) is usually sub-linear, slowing down as the system gets larger and more complex [12, 13]." As a result, Godfrey and Tu showed that Lehman's Laws and Turski's assumptions do not apply to Linux operating system kernel and more specifically, the growth rate is super-linear rather than sub-linear.

The first reason for Linux being an exception seems to be that at the time Lehman proposed his laws on software evolution, open source software was not so popular yet. But one question raised to us: is this just Linux, or all open source software do not obey Lehman's Laws? Therefore, we decided to extend the experiment to another well known open source project, Firefox. As we went through the life cycle of Firefox, we found some valuable information about its evolution. As for methodology, we decided to not only focus on the size of the source for Firefox but also to look at the project in more details such as different sub-systems and programming languages used. These detailed views also help us to gain some valuable information about Firefox's evolution.

2.3 Introduction of Firefox Project: Trunk-Branch structure of development

Firefox development team adopts a trunk-branch structure for their code. Most of the development activities happen on "trunk" of the code. And when the trunk is ready for the next stable release, its code will be "frozen" and a copy of it would be created and become a "branch". The branch is then handed over to testing and QA team for testing and defect-fixing. The next major release would be from the branch, whereas trunk will be unfrozen, and evolve towards next stable release.

This trunk-branch development structure brings promise to the evolution of the product, but also makes the maintenance of the product more difficult. The balance and trade-off between fast evolution, and ease of maintenance, is an important decision the project leader must make carefully.

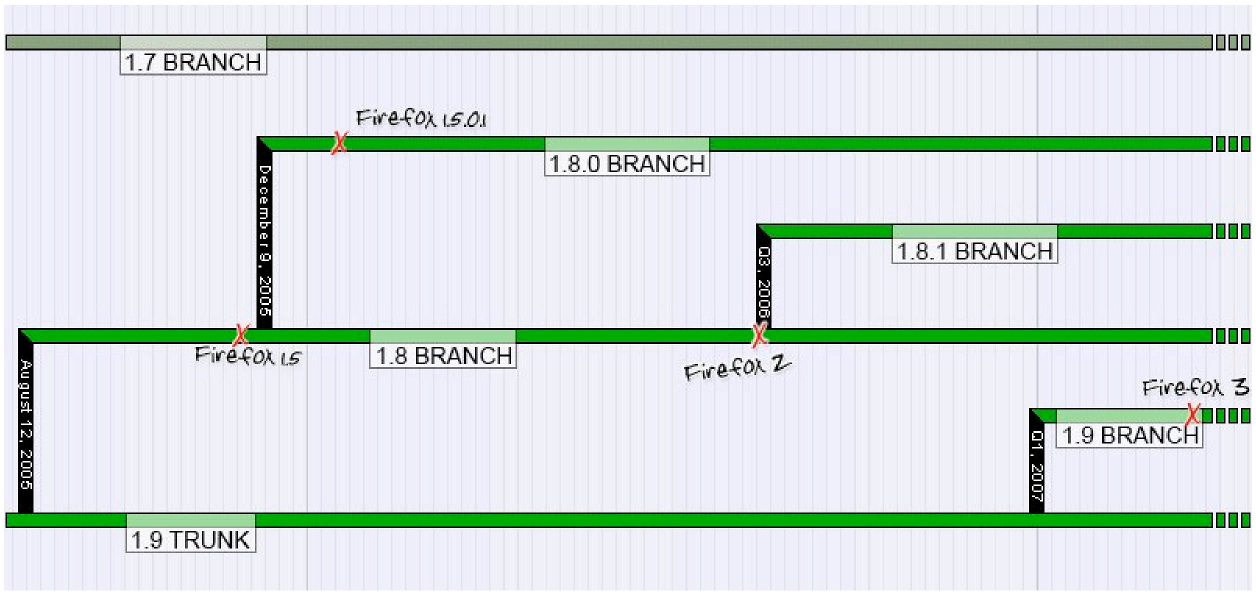


Figure 2-: Branch Mechanics of Firefox. [16]

Branches are numbered after Gecko version, the layout engine/core on which Firefox is based. Also note that the diagram is already outdated. But it still correctly represents the structure of Firefox code.

(Notice the difference between Gecko's version number and Firefox's version number: branches are numbered after the version number of Gecko)

Gecko is the layout engine on which Firefox is based. It evolved from v1.7 to v1.9 (current trunk) since 2004. It is not clear when Gecko v1.8 branched off from v1.7 trunk. (I'll be working on it, but it is not as important). v1.8 branch was created on Aug. 12, 2005, after the release of Firefox 1.1alpha2 on July 12 2005. (Notice that Firefox 1.1alpha and Firefox 1.4 later become Firefox 1.5).

On Dec. 09 2005, after the official release of Firefox 1.5, in order to better maintain Firefox 1.5.0.x releases, and for the convenience of the development of Firefox 2.0.0.x, a sub-branch was created: Firefox 1.5 were moved to v1.8.0.1 branch, and Firefox 2.0 development are based on v1.8.1 branch.

The trunk moved on to Gecko v1.9 till now. The life cycle of v1.7 branch (Firefox 1.0.x) ended on April 13, 2006 with the release of 1.0.8. Gecko v1.8 branch (Firefox 1.5.0.x)'s life cycle ended on May 30 2007 with the release of 1.5.0.12. The latest v1.8.1 branch release (Firefox 2.0.0.13) was released on March 25 2008.

3. Evolution of Firefox

3.1 Growth of Firefox

As mentioned in section 3, we decided to perform the study on Firefox using 3 approaches. First to compare different major releases and also major update patches in terms of total lines of codes. To make our job easier we got help from software called SLOCCount [15]. The result is shown is figure 3 below.

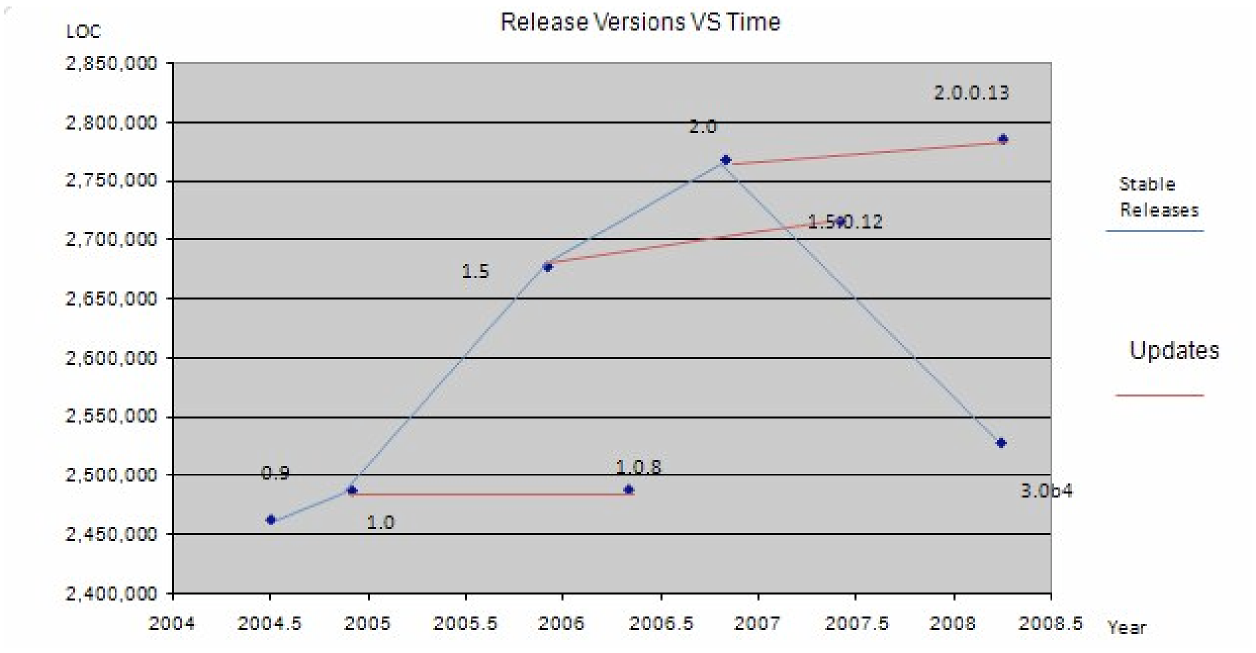


Figure 3-Growth in Firefox Releases

A major decrease in size can be seen from version 2.0 to version 3.0b4. This is because Mozilla team decided to discard one big module (mailnews) which is actually not part of Firefox in the source code package (the source package originally contained a large amount of code that is shared between different applications in the Mozilla Suite). In fact, if we add the component in 2.0 back to version 3.04b, the total size will be slightly larger than version 2.0’s size.

3.2 Architectural Change of Firefox

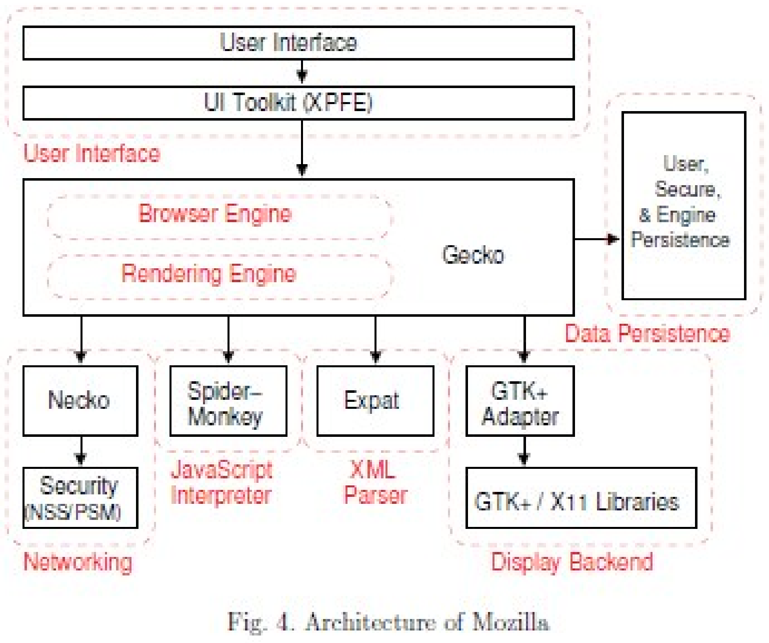


Figure 4 Architecture of Mozilla

Next, we decided to focus on the major dynamic subsystems in Firefox. The results are shown in figure three. We decided not to contain the mail-news subsystem in this graph since it existed with almost no changed in all the preliminary versions and was not included in version 3.0b4.

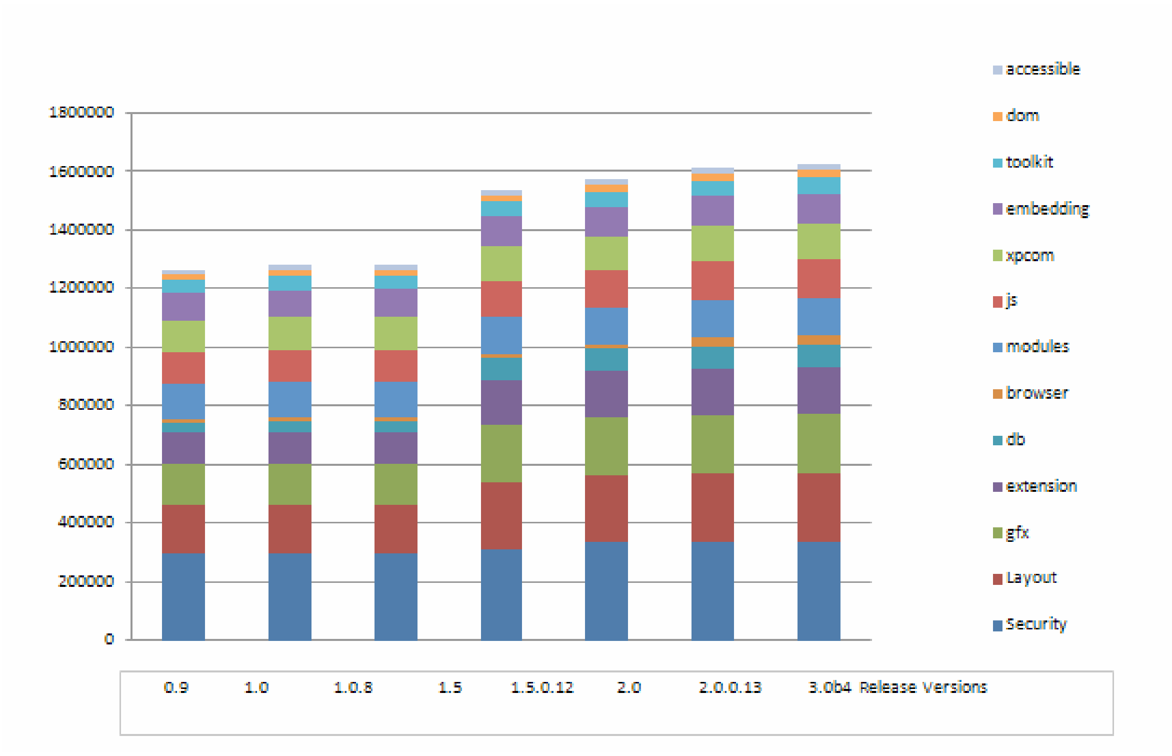


Figure 5-Major Dynamic Subsystems VS Release Version

One interesting we observed in the analysis is that although there are several directories kept growing during the development of the project, there are also some directories remain almost unchanged.

Security and accessibility are the two directories with constant and significant growth over time. This suggests that although the core function of a web browser stays unchanged, peripheral functions and features are always evolving, which seems endorse Lehman's observation on software evolution. DOM (Layout data structure for web pages) and JavaScript interpreter also grew constantly, which we believe is a result and reflection of the fast-changing web page layout standard.

On the other hand, almost all the core modules of the browser remain relatively stable. There is no significant growth in terms of SLOC for Gecko, the layout engine, which evolved from 1.7 to 1.9 over a 6-year span. Other components, including networking, parser, interface front-end, and display backend all stay relatively stable over time.

Also, the overall structure of the browser stayed almost unchanged, although we too observed some minor architectural change. And we noticed that this kind of architectural change is finished mainly through "copy and paste", which means the developers simply move all the files in one directory into another one, and then modify and build new code in the new location.

3.3 Programming Language

The last result we have driven is about the programming languages used. Figure 7 describes the major programming languages used in Firefox (The ones that contain more than 0.05 of the code). As can be easily seen that C++ is the most used languages but losing its importance and the second important one is Ansi-C. One of the reasons that C++ has a major decrease in the last version is again the omission of the mail-news component which was mainly written in CPP. Also, another change that we might not be able to see in the graph but we drove from the results was that, assembly is being used more and more in every new version and it is mainly used in the security module.

3.4 Release history and strategy of Firefox

For each major release of Firefox, there are typically two or three alpha releases, two or three beta releases, two or three release candidates, and eight to ten "dot-dot" releases (security fixes and updates) following the stable release. The interval between two security updates is usually between six weeks to eight weeks. (This is approximately the same or a bit longer than the frequency of the release of patches for Internet Explorer, which is usually once a month.)

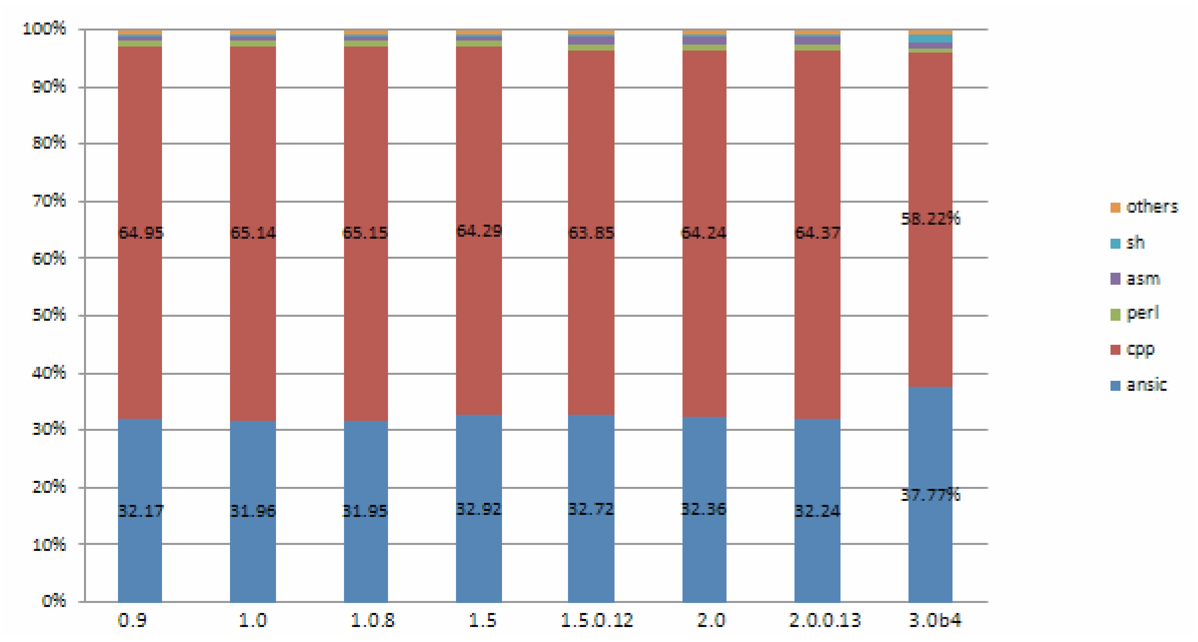


Figure 6 - Programming Languages VS. Different Versions

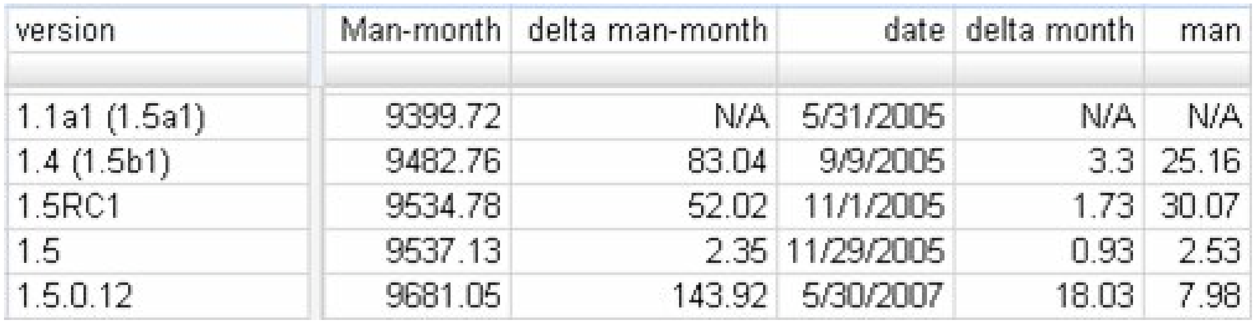
3.4 Constant Beta Phase?

One feature worth discussing about the release strategy of Firefox is that the definition of alpha/beta release is very different. Traditionally, alpha version refers to those that are only restricted to internal developers rather than public released, whereas beta version is released for public testing purpose. But because Firefox is open source software, its source code is always available online for free download. This breaks down the boundary between traditional alpha releases and beta releases. As defined in the roadmap of Firefox 2 (http://www.mozilla.org/projects/firefox/roadmap.html) alpha releases are feature-incomplete, whereas beta releases contains all the features to be shipped. Although the team warns that alpha releases are unstable and should not be used by general users, this kind of fast release strategy would potentially lead to differences in marketing of the product, which worth future study.

However, based on the data provided by SLOCCount, we observed a clear distinction between different development phases, particularly on the 1.8.1 branch on which Firefox 2.0 was built. Towards the end of the life cycle of the branch, the effort put into the development tend to decrease. Alpha phase is the one in which most development effort is put. And only a small portion of the total effort was put into the maintenance phase.

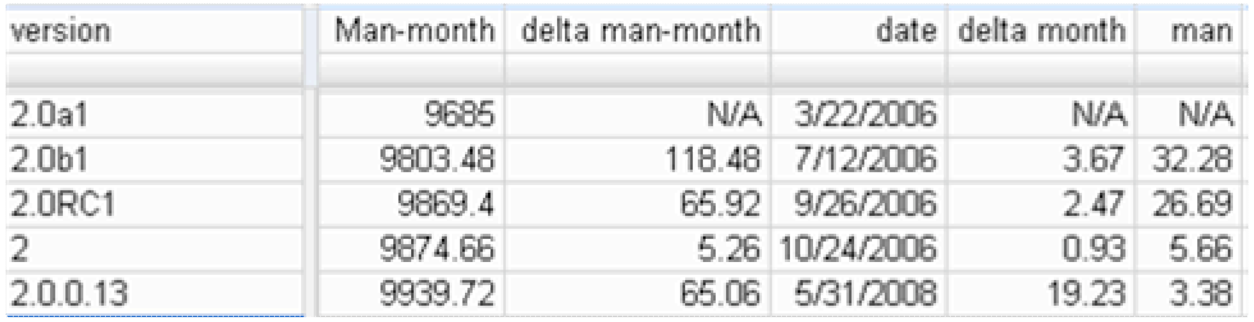
The pattern is not very clear on the 1.8.0 branch on which Firefox 1.5 was built, partly because when the branch was created, the development plan was not yet clearly drawn, as we could see from the confusion in its versioning mechanism: the version number of Firefox first jumped from 1.1 to 1.4, and 1.4 was first released as a official version, but soon it was decided that it should work as a beta version of the next stable release which would later be numbered as 1.5. It seems that Firefox 1.5 took more effort in the maintenance phase, indicating that it is a little buggy, comparing with well-accepted Firefox 2.0.

Unfortunately, the trunk-branch structure of the development make it very difficult to estimate the total effort based on source lines of code. As suggested by Prof. Godfrey, an estimation based on CVS commits (process-based) rather than on SLOC (entity-based) would be helpful, and would be an interesting topic for future study.



version	Man-month	delta man-month	date	delta month	man
1.1a1 (1.5a1)	9399.72	N/A	5/31/2005	N/A	N/A
1.4 (1.5b1)	9482.76	83.04	9/9/2005	3.3	25.16
1.5RC1	9534.78	52.02	11/1/2005	1.73	30.07
1.5	9537.13	2.35	11/29/2005	0.93	2.53
1.5.0.12	9681.05	143.92	5/30/2007	18.03	7.98

Table 1. Effort estimation of Firefox 1.5



version	Man-month	delta man-month	date	delta month	man
2.0a1	9685	N/A	3/22/2006	N/A	N/A
2.0b1	9803.48	118.48	7/12/2006	3.67	32.28
2.0RC1	9869.4	65.92	9/26/2006	2.47	26.69
2	9874.66	5.26	10/24/2006	0.93	5.66
2.0.0.13	9939.72	65.06	5/31/2008	19.23	3.38

Table 2. Effort estimation of Firefox 2

4. Firefox VS Lehman Laws

It is fairly easy to connect the evolution of Firefox to Lehman's first, second, fifth, sixth, and 8th laws and it appears to us that Firefox also obeys these laws. The different major releases together with patch updates indicate different stages of updates and bug fixes applied to Firefox. As can be seen in figures 2 and 3, different versions of Firefox change in terms of size and architecture, so Lehman's first law is also satisfied. Also his sixth law can be easily verified by figure 3.

The most interesting one of the laws that Godfrey and Tu questioned, is law #5. Graph 2 shows a decrease in amount of changes in newer version of Firefox which is in accordance to Truski's inverse square model. (Remember that this model did not apply to Linux operating system's kernel.)

We also talked about the average activity rate. As discussed, we observed a clear distinction between different development phases, particularly on the 1.8.1 branch on which Firefox 2.0 was built. Towards the end of the life cycle of the branch, the effort put into the development tend to decrease. Alpha phase is the one in which most development effort is put. And only a small portion of the total effort was put into the maintenance phase. This seems to be a reflection of what Lehman proposed as "conservation of organizational stability", although future study is needed to further clarify the definition and measurement of this statement.

Table 3 shows an improved version of Lehman's set of laws we would like to propose. In this version we have tried to fix the problems that cause Lehman's law inapplicable to some open source software systems. We also tried to combine some of Lehman's law in order to drive a more concrete description of the process of software evolution.

To talk about our patterns in greater details, first we decided to omit Lehman's 5th law for our list. We believe that a law is valid when there is no exception but as we know Godfrey and Tu already showed that Lehman's fifth law does not hold for their case study. . There might be several reasons, but we believe that the most important one is that Linux is a really large project which demands huge maintenance effort for its modules/subsystems. Also, an operating system kernel needs to get updated as the hardware technology improves; and we all know that the improvement rate in computer hardware technology has been really fast in the last 20 years. Another important reason can be that developers usually do not participate in open source project for financial payback. These projects are not supposed to bring direct profit for the owners either. Therefore, developers usually participate just for their own interest. Also they do not meet any deadlines, nor do they face any pressure for the tasks they are working on. These reasons in total cause a super linear growth in Linux Kernel. This also has direct relationship with Lehman's 7th law. Therefore we decided to not to include Lehman's 5th and 7th laws in our pattern. Instead we add a new pattern as our pattern #5. We claim that considering time, budget and other restrictions that might cause the systems face decrease in quality, it might get harder for the product to get changed.

To talk about our patterns in greater details, first we decided to omit Lehman's 5th law for our list. We believe that a law is valid when there is no exception but as we know Godfrey and Tu already showed that Lehman's fifth law does not hold for their case study. . There might be several reasons, but we believe that the most important one is that Linux is a really large project which demands huge maintenance effort for its modules/subsystems. Also, an operating system kernel needs to get updated as the hardware technology improves; and we all know that the improvement rate in computer hardware technology has been really fast in the last 20 years. Another important reason can be that developers usually do not participate in open source project for financial payback. These projects are not supposed to bring direct profit for the owners either.

No	Name	Statement
1	Continuing Change	A software system must be continually adopted else it becomes progressively less satisfactory in use
2	Continuing Growth	Software systems must be continually enhanced to maintain user satisfaction over system Lifetime
3	Subsystems evolutions	While the growth in the whole system might not be very visible, subsystems might face great changes in structure, size or implementation
4	Conservation of Organizational Stability	(Although it did not appear quite as expected , future analysis is needed in future because we are analyzing the data in a different manner)
5	Quality and complexity issues	Considering time, budget and other restrictions that might cause the systems face decrease in quality, it might get harder for the product to get changed.
6	Feedback	Software Evolution processes is multi-level, multi-agent, multi-loop feedback systems.

Table 3- Suggested Improved Laws for software evolution

Therefore, developers usually participate just for their own interest. Also they do not meet any deadlines, nor do they face any pressure for the tasks they are working on. These reasons in total cause a super linear growth in Linux Kernel. This also has direct relationship with Lehman's 7th law. Therefore we decided to not to include Lehman's 5th and 7th laws in our pattern. Instead we add a new pattern as our pattern #5. We claim that considering time, budget and other restrictions that might cause the systems face decrease in quality, it might get harder for the product to get changed.

We also noticed that in order to be able to talk about the process of evolution for a specific software project, it is insufficient to only examine its total size in terms of lines of code. Some components may become larger and some may get smaller or even disappear in new releases. This might cause a big change in the total size of the project, as shown in our results for Firefox 3.0b4, which could have mislead us. Therefore, as other researchers have also suggested [8], we add a new law to our table. Our law number 3 suggests that for studying the evolution of any large software system we need to consider the details as much as possible. Subsystems need to be considered. Different programming languages used can also be helpful to understand the total process of evolution.

Unfortunately, our experiment did not test Lehman's second law. Therefore we just did not talk about his second law in our collection of patterns.

5. Conclusion

In this paper, we extend the case study of Linux done by Godfrey and Tu to Firefox, another well-known open source project, and observed that the growth of Firefox is slowing down, which endorsed Lehman's Law of Software Evolution. We also discussed the trunk-branch structure and the "constant beta phase" strategy used by the project, and its impact on the project development in terms of change in architecture and effort. Finally we proposed possible modifications to Lehman's Law of Software Evolution.

References:

[1] M.M. Lehman (1978), Laws of Program Evolution–Rules and Tools for Programming Management, Proceedings of the Infotech State of the Art Conference, Why Software Projects Fail , London, England, April 9–11, 1978, pp. 1V1–lV25.

[2] Nazim H. Madhavji , Juan Fernandez-Ramil , Dewayne Perry .Software Evolution and Feedback: Theory and Practice, chapter 1

[3] Lehman MM, and Stenning V, FEAST/1: Case for Support, Department of Computing, Imperial College, London, UK, Mar. 1996. Available from the FEAST web site [5].

[4] Lehman MM, FEAST/2: Case for Support, Department of Computing, Imperial College, London, UK, Jul. 1998. Available from links at the FEAST project web site [5].

[5] FEAST Projects Web Site, Department of Computing, Imperial College, London, UK, http://wwwdse.doc.ic.ac.uk/~mml/feast/.

[6] M.M. Lehman (1978), Laws of Program Evolution–Rules and Tools for Programming Management, Proceedings of the Infotech State of the Art Conference, Why Software Projects Fail , London, England, April 9–11, 1978, pp. 1V1–lV25.

[7] *Evolution in Open Source Software: A Case Study*, Michael W. Godfrey and Qiang Tu, *Proc. of the 2000 Intl. Conf. on Software Maintenance* (ICSM-00), San Jose, October 2000.

[8] *Mining Large Software Compilations over Time: Another Perspective of Software Evolution*, Gregorio Robles, Jesús M. González-Barahona, Martin Michlmayr, Juan Jose Amor, *Proc of 2006 Workshop on Mining Software Repositories* (MSR-06), Shanghai, May 2006.

[9] L.A. Belady and M.M. Lehman (1972), An Introduction to Growth Dynamics, in W. Freiburger (ed.), StatisticalComputer Performance Evaluation, Academic Press, New York, pp. 503–511.

[10] M.M. Lehman (1982), Program Evolution, Symposium on Empirical Foundations of Computer and Information Sciences, 1982, Japan Information Center of Science and Technology, published in J. Info. Proc. And Management, 1984, Pergamon Press

[11] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski. Metrics and laws of software evolution—the nineties view. In *Proc. of the Fourth Intl. Software Metrics Symposium (Metrics'97)*, Albuquerque, NM, 1997.

[12] M. M. Lehman, D. E. Perry, and J. F. Ramil. Implications of evolution metrics on software maintenance. In *Proc. Of the 1998 Intl. Conf. on Software Maintenance (ICSM'98)*, Bethesda, Maryland, Nov 1998.

[13] W. M. Turski. Reference model for smooth growth of software systems. *IEEE Trans. on Software Engineering*, 22(8), Aug 1996.

[14] *Software Aging*, David Parnas, *Proc. of 1994 Intl. Conf. on Software Engineering* (ICSE-94), Sorrento, Italy, 1994.

[15] www.dwheeler.com/sloccount/

[16] http://www.mozilla.org/roadmap/branching-2005-12-16.png, accessed November 30, 2007

[17] Adapted from *A Case Study in Architectural Analysis: The Evolution of the Modern Web Browser,*A Grosskurth, MW Godfrey

http://plg2.math.uwaterloo.ca/~migod/papers/2007/emse-browserRefArch.pdf