View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by PDXScholar

Portland State University
PDXScholar

Engineering and Technology Management Student Projects

Engineering and Technology Management

Spring 2018

Technological Evolution in Software Engineering

Cody Miller
Portland State University

Let us know how access to this document benefits you.

Follow this and additional works at: https://pdxscholar.library.pdx.edu/etm_studentprojects

Part of the Software Engineering Commons, and the Technology and Innovation Commons

Citation Details

Miller, Cody, "Technological Evolution in Software Engineering" (2018). *Engineering and Technology Management Student Projects*. 2239.
https://pdxscholar.library.pdx.edu/etm_studentprojects/2239

This Project is brought to you for free and open access. It has been accepted for inclusion in Engineering and Technology Management Student Projects by an authorized administrator of PDXScholar. For more information, please contact pdxscholar@pdx.edu.

Technological Evolution in Software Engineering

Course Title: Management of Tech Innovation
Course Number: 549
Instructor: Dr. Weber
Term: Spring
Year: 2018
Author(s): Cody Miller

ETM OFFICE USE ONLY	
Report No.:	
Type:	Student Project
Note:	

Contents

Abstract 3
Review of Literature 4
  Programs, Life Cycles, and Laws of Software Evolution 4
    Program Maintenance 4
    The Life Cycle 5
    Evolution 6
    Analysis & Findings 8
  Software Maintenance and Evolution: A Roadmap 8
    Software Maintenance 9
    A Staged Model for Software Lifecycle 10
    Analysis & Findings 13
  The Linux kernel as a case study in software evolution 13
    Research Data 14
    Evolution of the Linux Kernel 14
    Analysis & Findings 18
Conclusion 18
References 20

Abstract

In all software development processes, the software must evolve in response to its environment or user needs to maintain satisfactory performance [1]. If software doesn't support change, it gradually becomes useless [2]. With many organizations today, being software-centric organizations, this has huge implications for their business: evolve your software, or risk your software becoming gradually useless, and therefore, your entire business.

Technology Evolution is a highly relevant subject, Intel's business model for the last 50 years, has been that of Moore's Law [3], a hardware centric Technology Evolution model. As a Software Engineer at Intel, our business group faces a similar issue, we must continually adapt, and evolve our software, in response to our customer's needs, and current technology trends, if we don't evolve our software, our competitors will evolve theirs faster, and our business group, will gradually cease to exist, without competitive, and evolving software.

The software evolution phenomenon was first identified in the late 60s [4] though not termed as such till 1974 [5]. The goal of this article, is to explore the current literature on software evolution, and its impacts on software development activities, and software organizations. As a manager, and practicing Software Engineer, software evolvability, the ability, inter alia, for responsiveness and timely implementation of needed changes, will play an ever increasing more critical role in ensuring the survival of a society ever more dependent on computers [4].

Review of Literature

Programs, Life Cycles, and Laws of Software Evolution

This article identifies the sources of evolutionary pressure on computer applications(software) and shows why this results in a process of never ending maintenance activity.

In 1977, the total U.S. expenditure on programming is estimated to be between 50-100 billion, this represents more than 3 percent of the U.S. GNP [2]. These ever-increasing numbers, and the rise of "Silicon Valley", one could argue that software effectiveness, and software evolution, is a significant component of national economic health. As software plays an ever-increasing role in society, it becomes more critical to create and maintain effective, cost-effective, and timely software. For more than two decades however, the programming fraternity, and through them the computer user community, has faced serious problems in achieving this [6].

Program Maintenance

Of the 50 – 100 billion U.S. expenditure on software in 1977, some 70 percent was spent on program *maintenance* and only about 30 percent on program *development*. This ratio is generally accepted by the software community as characteristic of the state of the art [2]. To put this in perspective, software maintenance can be classified as all changes made to a software application, after its first installation. Whereas in hardware systems, large changes to hardware typically result in a redesign, retooling, and construction of a new model. With software, improvements, and adaptations to the ever-changing environment, can be achieved by alterations, deletions, and extensions of the existing code base. Typically, new capabilities, not recognized during the development

phase, is added onto the existing structure, without an entire system redesign, this is what makes software maintenance so critical, to both software development, and in turn, software evolution.

The Life Cycle

The dynamic, evolutionary nature of computer applications, of the software that implements them and of the process that produces both, has in recent years given rise to a concept of a program life cycle and to techniques for life-cycle management [2]. In *Figure 1,* Boehm describes the Software Lifecycle, the first 3 phases (Systems Requirements, Software Requirements, and Preliminary Design), occur during the "definition" phase of software development. The "implementation" phase (Detailed Design, Code & Debug, and Test) make up the next set of software development activities. Lastly, we have the "maintenance" phase, which consumes some 70 percent of software development costs, and makes up a key phase of software evolution.

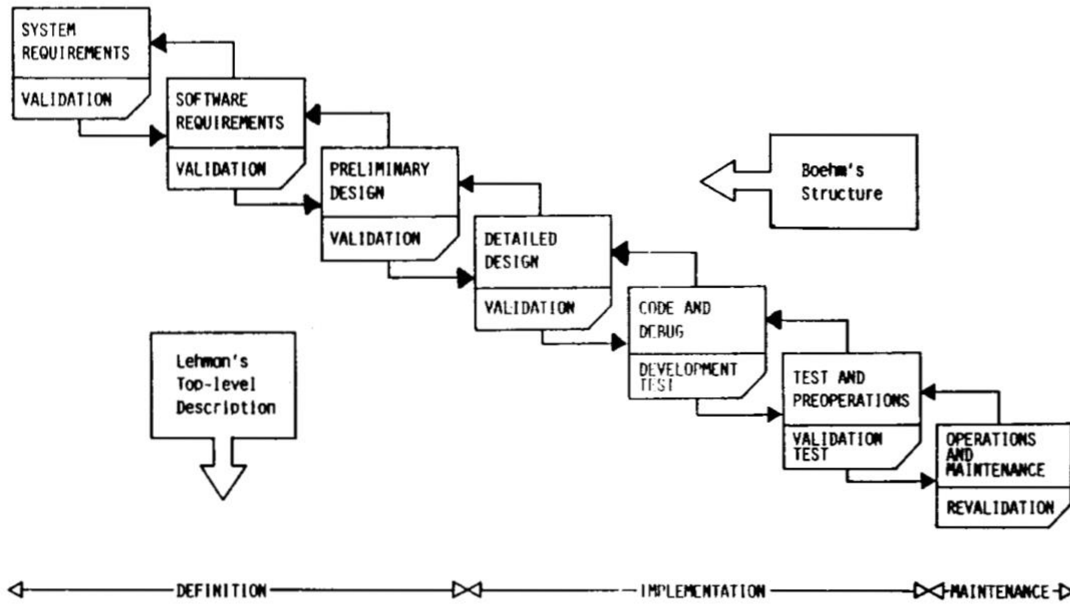

Figure 1 - The Software Life Cycle according to Boehm

In studying program evolution, repetitive phenomena that define a life cycle (Figure 1), can be observed on different time scales, representing various levels of abstraction. The highest-level concerns *successive generations* of system sequences. Each generation is represented by a sequence of system releases. This level corresponds most closely to that found in the more general systems situation, with each generation having a lifespan of from, say, five to twenty years [2]. Due to these lifespans of 5-20 years, it can be difficult for individuals to observe this evolution, measure its dynamics, and model it as a life cycle process.

Evolution

As shown in *Figure 1,* you can see how software evolution, correlates with the software life cycle, indicating that software evolution is an intrinsic, feedback driven, property of software. The resultant evolution of software appears to be driven and controlled by human decision, managerial edict, and programmer judgement [2].

In this article, Lehman describes 5 laws, of Software Evolution [2]. These laws are not just descriptions of the evolutionary process, these laws also represent principles in software engineering. It is important to note, these laws are not like laws of physics, or chemistry, they are not immutable, instead, they arise from the habits and practices of people and organizations. The laws, therefore form an environment within which the effectiveness of programming methodologies and management strategies and techniques can be evaluated.

1. **Continuing Change**: A program that is used and that as an implementation of its specification reflects some other reality, undergoes continual change or becomes progressively less useful. The change or decay process continues until it is judged more cost effective to replace the system with a recreated version.
2. **Increasing Complexity**: As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it.
3. **The Fundamental Law of Program Evolution**: Program evolution is subject to a dynamic which makes the programming process, and hence measures of global project and system attributes, self-regulating with statistically determinable trends and invariances.
4. **Conservation of Organizational Stability**: During the active life of a program the global activity rate in a programming project is statistically invariant.

5. **Conservation of Familiarity**: During the active life of a program the release content (changes, additions, deletions) of the successive releases of an evolving program is statistically invariant.

Analysis & Findings

This article rationalizes the widely held view, first expressed in Garmisch [7], that there is an urgent need for a discipline of software engineering. This should enable the cost-effective planning, design, construction, and maintenance of effective programs that provide, and then continue to provide, valid solutions to stated (possibly changing) problems, or satisfactory implementations of (possibly changing) computer applications [2].

Secondly, in this article, we recognized how software evolves, through the software lifecycle, with a large majority of evolutionary activities, and costs, occurring in the maintenance phase of the software life cycle. Understanding that software changes, and evolves over time, 5 laws, that appear to govern the dynamics of the evolution process was introduced [2].

Software Maintenance and Evolution: A Roadmap

Software maintenance and evolution are characterized by their huge cost and slow speed of implementation. Yet they are inevitable activities – almost all software that is useful and successful stimulates user-generated requests for change and improvements [8]. The objective of this article, is to describe the current landscape for research into software maintenance and evolution over the next 10 years, with the aim of improving the speed and accuracy of change while reducing cost. This is accomplished through two

approaches: A new model of software evolution called the *staged model* is proposed. Second, a longer term, and much more radical vision of software evolution is presented [8].

Software evolution lacks a standard definition, with some researchers and practitioners using it as a substitute for maintenance. In this article, the authors refer to maintenance as general post-delivery activities, and evolution to refer to a particular phase, in the staged model.

Software Maintenance

Software maintenance is defined in IEEE93 as "The modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment". ISO95, describes a similar definition, "The software product undergoes modification to code and associated documentation due to a problem or the need for improvement. The objective is to modify the existing software product while preserving its integrity". The key point here, is that both of these definitions, show a post-delivery nature.

A widely cited survey [9], and repeated by others in different domains, pointed to a very high expenditure of life-cycle costs in the maintenance stage. Lientz and Swanson further categorized the maintenance phase into four classes [9]

- **Adaptive** – Changes in the software environment
- **Perfective** – new user requirements
- **Corrective** – Fixing errors
- **Preventative** – Prevent problems in the future

Of these 4 maintenance classes, the survey showed that around 75% of maintenance effort was spent on the first two classes, many subsequent studies, suggest a similar problem [8]. What this study tells us, is that the incorporation of *new user requirements* is the core problem for software evolution and maintenance.

There are several implications to these results. If changes can be anticipated during design time, they can be built in by some form of parameterization. The problem is, many changes actually required, are ones that designers cannot even conceive. This makes software maintenance important for two reasons:

1. It consumes a large portion of the overall lifecycle costs
2. The inability to change software quickly and reliably means that business opportunities are lost

A Staged Model for Software Lifecycle

The conventional lifecycle model, as described in *Figure 1,* is no longer useful for modern software development. It does not help with reasoning about component-based systems, and does not help with planning software evolution [8]. The staged model of software lifecycle was introduced in [10], and is summarized in *Figure 2.*

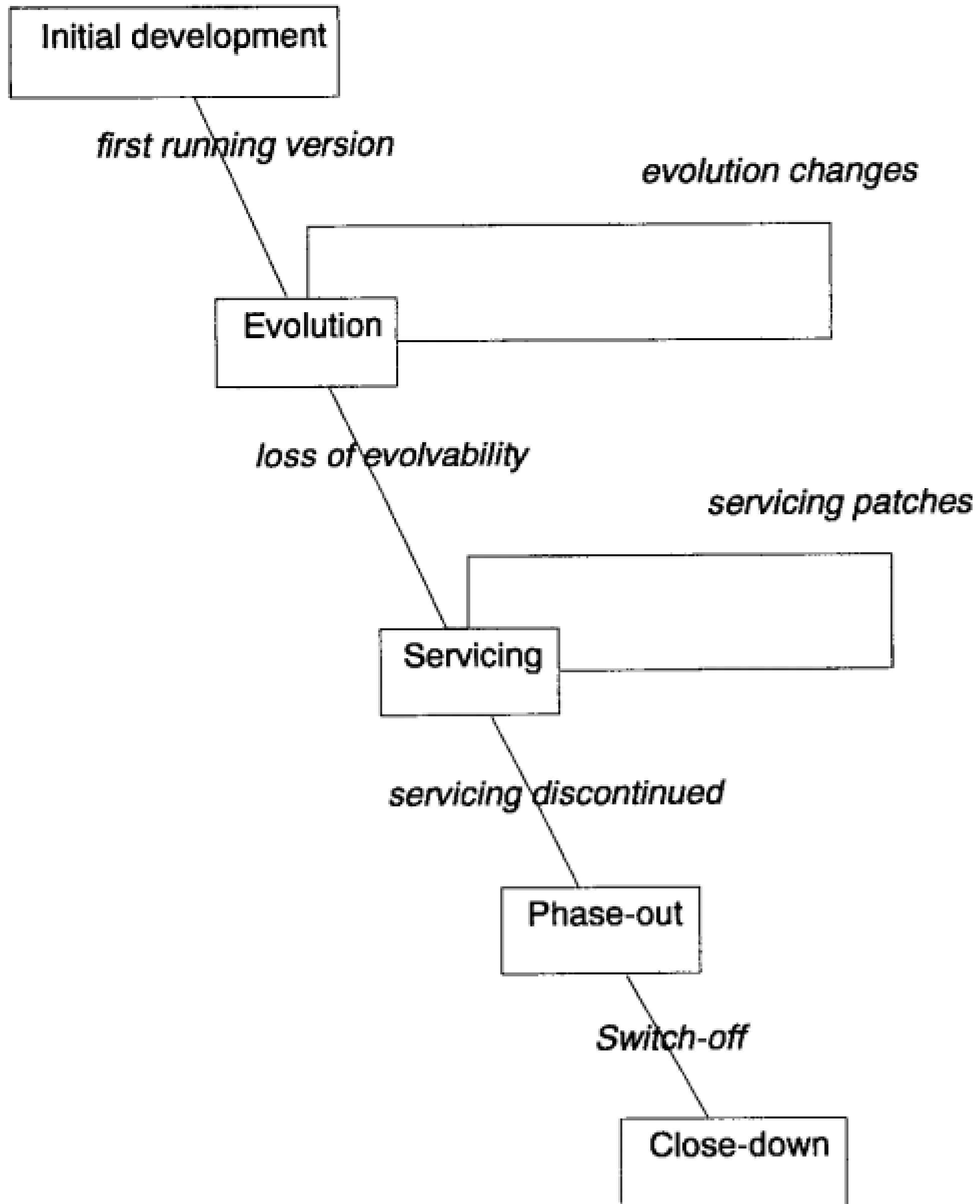

Figure 2 - The Simple Staged Model

In the Simple Staged Model, the software lifecycle is represented as a sequence of stages, "Initial Development" is the first stage, as shown in *Figure 2.* The primary difference in this model, is the "maintenance" phase is separated into an "evolution" stage, proceeded by "servicing, and "phase-out" stages.

The Initial Development phase, is when the 1st version of a software application is developed. It does not need to have full features, but what will remain constant is that it has architecture that will remain throughout the staged model. An important outcome of this stage is the knowledge the programming team acquires, this knowledge is a crucial prerequisite for the subsequent stage of evolution [8].

The Evolution stage, begins when the Initial Development stage is complete. The goal of the Evolution stage, is to adapt the software to the ever-changing user requirements and environment. The Evolution stage is also responsible for correcting software bugs, and responds to developer, and user learning.

In business terms, the software evolves, because it is successful in the marketplace, revenue streams are buoyant, user demand is strong, the development atmosphere vibrant and positive, and the organization is supportive. Return on investment is excellent [8].

Architecture, and team knowledge make software evolution possible. This allows the team to make changes in the software, without damaging the architecture. When one of these aspects disappears, the program is no longer evolvable, and enters the "Servicing" stage, as shown in *Figure 2*. The Servicing stage, consists of very small changes, such as, patches, and wrappers. From a business perspective, this occurs typically when the software is no longer a core product to the business, and the cost-benefit of changes, are much more marginal [8].

The two final stages, Phase-Out, and Close-Down. A phase-out, means the Servicing stage has shut down, and users work around known software bugs, but the

software may still be in production. The Close-Down stage is when the software is shut down, and users are directed towards a replacement.

Analysis & Findings

A new software lifecycle model is proposed in this article (*Figure 2*), called the Staged Model. During Initial Development of the software, the main goal of this stage is to ensure evolution can be enabled easily, through successful knowledge transfer. This is an organization issue, and technological problem.

Software evolution needs to be addressed as a business issue as well as a technology issue, and therefore is fundamentally interdisciplinary [8]. In order to make progress, one must understand what evolution is, and how to start the process. In a strategic sense, advancement in software architectures is a critical piece to master.

Being able to change and evolve software easily, is a difficult challenge in the domain of Software Engineering. Change is a constant in software, and one of the main benefits of software, it is naïve to not think software evolution will be needed in the future. We can expect software evolution to be positioned at the center of software engineering [8].

The Linux kernel as a case study in software evolution

In this case study on software evolution, the researchers look at 810 versions of the Linux kernel, released over a period of 14 years, to understand the software's evolution, with Lehman's Laws of software evolution as a basis. The objective, being to understand if Lehman's laws of software evolution, are reflected in the development of the Linux kernel.

The Linux kernel is a large, and long-lived software system, in use throughout the world. Since version 1.0 was released in 1994, more than 800 releases have been made since. It's an open source project, providing data for this study of software evolution of a nature and scale that is impossible with other, especially closed-source, systems [11].

Research Data

All Linux releases from March 1994 to August 2008 were analyzed. This amounted to 810 releases, with 144 production versions, and 429 development versions. The entire source code was included in the analysis, including .h, and .c files. Both development, and product versions were included, as most of the evolution occurs in development versions [12].

Evolution of the Linux Kernel

Law 1: Continuing Change

To refresh, this law states that a program that is used must be continually adapted to changes in its usage environment, else it becomes progressively less satisfactory. Linux software must adapt to the changing hardware environment, these changes can be tracked to the arch subdirectory of the kernel, whereas driver changes pertaining to new peripheral support, can be tracked in the driver's directory. One can conclude, that code added to these two directories, reflects adaptation to the systems changing hardware environment [11]. A plot of how these two subdirectories grow over-time, is shown in *Figure 3* [11].

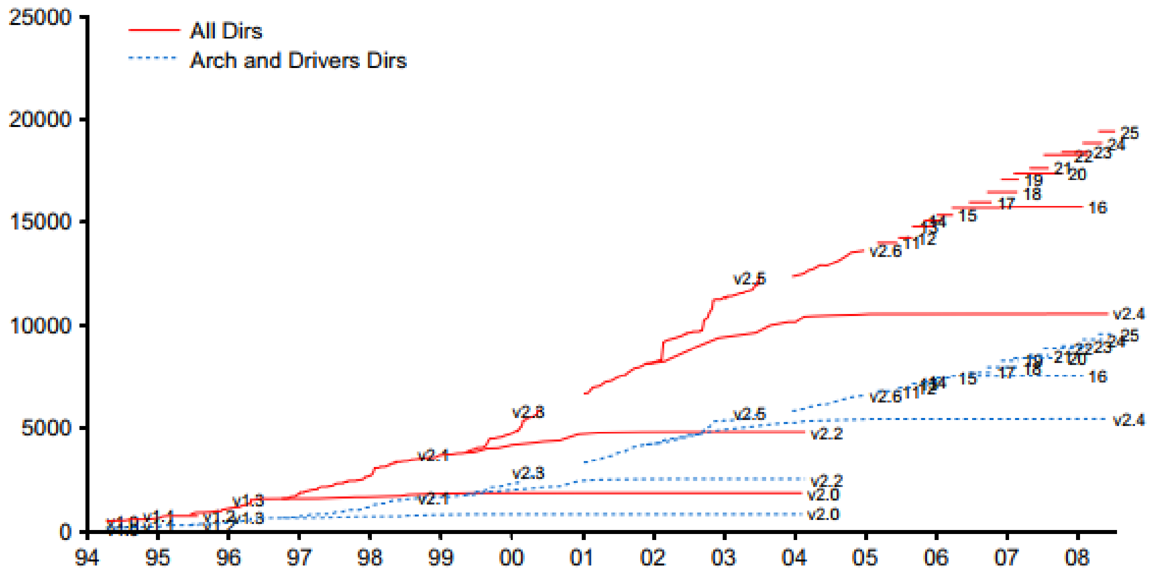

Figure 3 - The growth of source files in the arch and drivers directories as a fraction of the whole Linux system

As you can see from this chart, these two subdirectories grow over-time, mirroring the growth of the Linux kernel as a whole. Using this data, we can therefore assert that Linux exhibits continued change and adaptation to its environment, in accordance with Lehman's first law. While the original law is probably of wider scope, then this specific example [11].

Law 2: Increasing Complexity

Lehman's second law, as a program evolves its complexity increases, unless work is done to maintain it. This is a difficult law, to prove, or disprove formally, as it allows for both trends [11]. Lehman supports this law, through rationalization, asserting that adding features and devices becomes more complex. Given current Linux data, regarding the Linux code growth (not necessarily a good metric to measure this), does not display an inverse-square pattern, as claimed by Lehman.

Law 3: Self-Regulation

Lehman's third law, asserts that software evolution process is self-regulating, leading to a steady trend. The existence of self-regulation in the Linux kernel, may be established by observing growth trends, as shown in *Figure 4* [11].

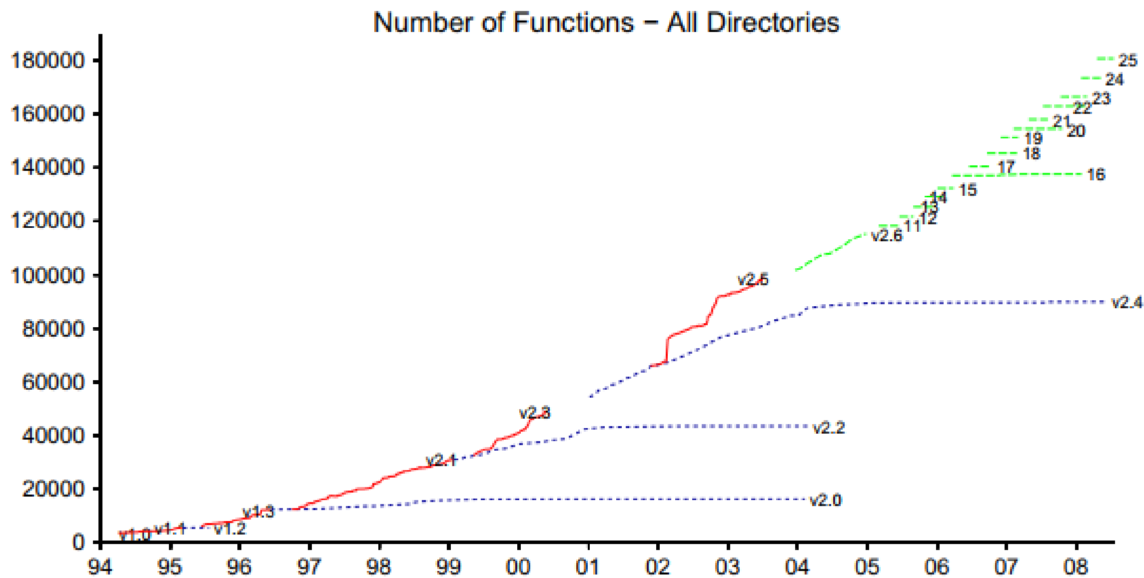

Figure 4 - The growth of Linux as measured by LOC and number of functions

As shown in Figure 4, exhibited growth patterns are steady, they do exhibit slight variations, that may be described as a ripple. The data also exhibits, the occasional larger jumps as a result of integrating a new subsystem, that was developed externally. The smooth growth rate, may be interpreted as resulting from self-regulation, but it may also be the result of an invariant work rate [11].

Law 4: Conservation of Organizational Stability

In this law, Lehman states the average effective global rate of activity on an evolving system is invariant over the life of the product. This is difficult to measure since we wish to observe "work" on the actual project.

Taken at face value, this law is patently false for Linux. The number of people contributing to Linux development has grown significantly over the years, and several companies now have dedicated employees assigned to working on them [11].

Law 5: Conservation of familiarity

According to this law the change between successive releases is limited, to allow developers to maintain understanding of the code base, and to allow users to still understand the system. Findings from the Linux kernel indicate that developers familiar with one version may expect little change in the subsequent ones, this holds true where the system grew significantly as well, because such additions are on existing known architectures, the knowledge base remains similar. However, when looking at changes between successive major versions, there is significant changes, with many users opting to stay on older versions, to maintain familiarity, and compatibility.

Thus, we have both support for this law (as witnessed by the longevity of production versions) and contradiction of the law (because successive production versions with significant changes are nevertheless released) [11]. *Table 1* [11], summarizes these findings, into Lehman's laws, and their manifestation in Linux.

Table 1 - Support for Lehman's Laws in Linux

No.	Lehman's Law	Manifestation in Linux
1	Continuing change	The arch and drivers directories, which account for 50–60% of the codebase, grow with the rest of the kernel, reflecting continued adaptation to the hardware environment
2	Increasing complexity (unless prevented)	While overall complexity grows with the code, the average per function is declining; in specific instances work to reduce complexity is evident
3	Self-regulation	Possibly supported by steady overall growth rates and fluctuation of incremental growth, but there is no direct support for a regulation mechanism
4	Conservation of organizational stability (invariant work rate)	Rate of releases has been relatively stable from 1997 to 2003. The 2.6 method of timed releases also creates an invariant amount of releases per time unit
5	Conservation of familiarity	Long-lived production versions reflect this law – successive minor releases have little functionality changes. But there are big changes between successive production versions

Analysis & Findings

The study presented here, was based on Lehman's Laws of software evolution. Obvious support was found for continuing growth and change, and invariant work rate. Conservation of familiarity seems to be combined with large changes when new Linux production versions are released. Preventative maintenance practices, seems to support the increasing complexity law. The hardest law to measure, and justify, is the self-regulating law, for which only anecdotal evidence is found.

Taking a global view of Linux's evolution, the authors find it to be a prime example of perpetual development – a system that is developed continuously in collaboration with its users, without elaborate specifications and planning [11]. These results are specific to Linux; however, some observations may generalize to other software systems as well.

Useful future research, would be to perform a similar study on a closed-source system, in order to understand the differences in development, the problem being, finding suitable data for such a study.

Conclusion

In this literature review, my research focused on 3 primary sources of literature [2], [11] [8]. A wealth of accompanying literature was reviewed, in addition to these articles, to gain a broad perspective, on the current state of Technology Evolution within Software Engineering. Topics ranged from Moore's Law, to software engineering economics, with the commonality of Technology Evolution, and Software Engineering [1], [3], [13]–[19].

After reading all of this literature, there were several trends observed, with respect to Technological Evolution in Software Engineering, and most importantly, findings that should help practicing managers, and leaders, to evolve, and better understand their software.

Two competing Software Development Lifecycles were proposed, the Software Development Lifecycle as proposed by Boehm, as shown in *Figure 1*, and the Simple Staged Model, as shown in *Figure 2*. While these models differed, there were many commonalties shared between the two, and some differences.

Software maintenance is a critical phase in software development activities, and the resultant evolution of the software. Not only is this a very important stage in software development, and its subsequent evolution, it's the costliest, and time-consuming activity of all software activity. Across all literature, this has been a common trend, with little, to no disagreements on the importance, and cost of software maintenance, and its role in software evolution.

I can also back this claim up, through my own professional experience as a Software Engineer at Intel. Our business group puts the majority of our software engineering resources into this "software maintenance" area, and I can see how it ties into the literature. From a business perspective, its always a battle, to determine where to allocate your resources, the maintenance phase, or the or the definition phase (defining future software, to eventually replace the software going through iterative maintenance, and evolution).

To be a successful software engineering manager, it's important to understand Technology Evolution, and its effects on software development activities. The software

engineering landscape is a complex landscape, and better understanding concepts such as the software development lifecycle (SDLC), Lehman's Laws of Software Evolution, and the importance of software maintenance on innovation, and its impact on the evolution of software, gives the practicing manager a "toolkit" to successfully navigate this ever-changing environment.

Technology Evolution, and its effects on Software Evolution, is not just some buzz-word, or academic only concept, it's a phenomenon that is occurring on a daily basis, across all facets of software engineering, some may recognize it, others may not. The ones that do, have a significant competitive advantage over others that don't. There is one thing that's certain in today's business environment: evolve your software, or risk your software becoming gradually useless, and therefore, your entire business going out.

References

[1] S. Murer, C. Worms, and F. J. Furrer, "Managed Evolution," *Informatik-Spektrum*, vol. 31, no. 6, pp. 537–547, Dec. 2008.

[2] M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proc. IEEE*, vol. 68, no. 9, pp. 1060–1076, 1980.

[3] E. Mollick, "Establishing Moore's Law," *IEEE Ann. Hist. Comput.*, vol. 28, no. 3, pp. 62–75, Jul. 2006.

[4] M. M. Lehman and J. F. Ramil, "Software evolution—Background, theory, practice," *Inf. Process. Lett.*, vol. 88, no. 1–2, pp. 33–44, Oct. 2003.

[5] M. M. Lehman, "Programs, Cities, Students— Limits to Growth?," in *Programming Methodology*, New York, NY: Springer New York, 1978, pp. 42–

69.

[6] E. J. Goldberg, “High Cost of Software,” *(Naval Post-grad. Sch. Monterey, CA)*, p. 138 pp, 1973.

[7] I. NATO Conference on Software Engineering Techniques (1969 : Rome, J. N. Buxton, 1936- Brian Randell, and N. S. Committee., “Software engineering techniques : report on a conference sponsored by the NATO Science Committee, Rome, Italy, 27th to 31st October 1969.”

[8] K. Bennett, V. R. the F. of S. Engineering, and undefined 2000, “Software maintenance and evolution: a roadmap,” *dl.acm.org*.

[9] B. P. Lientz and A. E. Burton, Swanson, *Software maintenance management : a study of the maintenance of computer application software in 487 data processing organizations*. 1980.

[10] K. H. Bennett, V. T. Rajlich, and N. Wilde, “Software Evolution and the Staged Model of the Software Lifecycle,” 2002, pp. 1–54.

[11] A. Israeli and D. G. Feitelson, “The Linux kernel as a case study in software evolution,” *J. Syst. Softw.*, vol. 83, no. 3, pp. 485–501, Mar. 2010.

[12] Thomas and L. Gray, “An analysis of software quality and maintainability metrics with an application to a longitudinal study of the linux kernel,” 2008.

[13] B. Boehm, *Software engineering economics*. 1981.

[14] R. Garud and M. A. Rappa, “A Socio-Cognitive Model of Technology Evolution: The Case of Cochlear Implants,” *Organ. Sci.*, vol. 5, no. 3, pp. 344–362, Aug. 1994.

[15] R. Kapoor and P. J. McGrath, “Unmasking the interplay between technology

evolution and R&D collaboration: Evidence from the global semiconductor manufacturing industry, 1990–2010," *Res. Policy*, vol. 43, no. 3, pp. 555–569, Apr. 2014.

[16] "Software technology evolution helps streamline offshore engineering: EBSCOhost."

[17] R. P. de Oliveira and E. S. de Almeida, "Evaluating Lehman's Laws of Software Evolution for Software Product Lines," *IEEE Softw.*, vol. 33, no. 3, pp. 90–93, May 2016.

[18] Y. Dong, M. Candidate, and S. Mohsen, "Does Firefox obey Lehman's Laws of software Evolution?"

[19] M. W. Godfrey and D. M. German, "On the Evolution of Lehman's Laws," *J. Softw. Evol. Proc*, vol. 0000, no. 00, pp. 1–7.