Journal of Software: Evolution and Process

JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
*J. Softw. Evol. and Proc.* 2014; **26**:613–619
Published online 15 November 2013 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/smr.1636

# On the evolution of Lehman's Laws

Michael W. Godfrey[1,*,†] and Daniel M. German[2]

[1]*School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada*
[2]*Department of Computer Science, University of Victoria, Victoria, British Columbia, Canada*

SUMMARY

In this brief paper, we honor the contributions of the late Prof. Manny Lehman to the study of software evolution. We do so by means of a kind of evolutionary case study: First, we discuss his background in engineering and explore how this helped to shape his views on software systems and their development; next, we discuss the laws of software evolution that he postulated based on his industrial experiences; and finally, we examine how the nature of software systems and their development are undergoing radical change, and we consider what this means for future evolutionary studies of software. Copyright © 2013 John Wiley & Sons, Ltd.

Received 5 April 2013; Revised 23 August 2013; Accepted 2 September 2013

KEY WORDS: software evolution; Lehman's Laws; software architecture

## 1. LEHMAN'S INTELLECTUAL JOURNEY

Meir 'Manny' Lehman did not follow a traditional career path for an academic. His father died when he was young, and Lehman had to enter the workforce to help support his family instead of attending a university. He got his first job in 1941 'performing maintenance on' (i.e., repairing) civilian radios in England at the height of the World War II, this was a job of real importance. For the most part, his work involved replacing the components that the 'tester' (or 'debugger', in software engineering parlance) had determined to be problematic. He found the work repetitive and dull. He decided that he really wanted to be a tester. One day, his foreman called in sick and Lehman was allowed to do testing, but when the foreman returned, Lehman was told to go back to maintenance. When Lehman protested that he thought he had been promoted, the foreman replied 'Well, you're not paid to think' [1]. Lehman would later dedicate a large portion of his life to demonstrate that those who do 'maintenance' of software should be paid to think, too. Lehman later attended Imperial College London where he received a PhD in 1957.

Like many other pioneers of computer science, Lehman lived the computer revolution from its inception; his early work was dedicated to building some of the first modern computers, and by the end of his life, he was witnessing the ubiquity of mobile computing. It is likely that he would have spent his life working on hardware, had it not been for the few years he spent at IBM between 1964 and 1972. IBM had originally hired him to help build physical computers, but in 1968, in a radical change of direction, he was asked to investigate programming practices within the company. This project took him to study the development of the landmark operating system IBM S/360 and its successor, IBM S/370. Lehman discovered that programmers were becoming increasingly interested in assessing their productivity, which they measured in terms of daily source lines of code and

*Correspondence to: Michael W. Godfrey, Computer Science, University of Waterloo, Waterloo, Ontario, Canada.
†E-mail: migod@uwaterloo.ca

passing unit-tests. He noticed that productivity was indeed increasing according to these measures, but at the same time the developers appeared to be losing sight of the overall product. In his words, 'the gross productivity for the project as a whole, and particularly the gross productivity as measured over the lifetime of the software product had probably gone down' [1]. It was during this period that he developed a close friendship with fellow IBM employee, Laszlo Belady. Together, they would challenge the prevailing models and assumptions of software maintenance processes, and champion the study of software evolution as a field in its own right.

In 1972, Lehman left IBM to join Imperial College London, where he would continue his work in software engineering research. Lehman was also instrumental in creating, at Imperial College, one the first academic programs in software engineering. Although he did not consider himself a programmer, his work at IBM had allowed him to study and understand programmers and their products better than most. He had witnessed first-hand the challenges of producing industrial programs for a real-world environment. And, most importantly, he noticed that the processes involved in developing and maintaining software appeared to form a kind of feedback system, where the environment provided a signal that had profound impact upon the continued evolution of the system.

Lehman's engineering-influenced views on software systems and their development were in stark contrast to some other well known computer scientists of the time, such as Edsger Dijkstra. For Dijkstra, a program was essentially a passive, mathematical entity that should be derived, iteratively, from a formal statement of what it was supposed to do. In this model, the software developer starts with a precise specification of the desired functionality, and then implements a series of formal 'step-wise' refinements, gradually making it more concrete, and ultimately executable; Dijkstra thus championed the view that teaching programming should emphasize creating a correct specification first, and then progressively transforming it through correctness-preserving transformations into a program that satisfies the specification [2]. Although Lehman recognized the value of Dijkstra's position, at the same time, he felt that it was an impractical model for the problem space of industrial software and for the style of development he had observed at IBM. Lehman postulated that programs could be divided into two main categories: S-type programs, which are derived from a rigorous specification that can be stated up-front, and can be proven correct if required; and E-type (evolutionary) programs, whose requirements are strongly affected by their environment and must be adaptable to changing needs [3]. In his view, E-type systems are those that are *embedded* in the real world, implicitly suggesting that S-type programs were less common—and thus less important—outside of the research world.

## 2. LEHMAN'S LAWS OF EVOLUTION

Lehman's best known work concerns his laws of software evolution, which he devised and refined with several collaborators—most notably, Laszlo Belady—over many years. In Lehman's view, 'The moment you install that program, the environment changes.' Hence, a program that is expected to operate in the real world cannot be fully specified for two reasons: it is impossible to anticipate all of the complexities of the real world environment in which it will run; and, equally importantly, the program will affect the environment the moment it starts being used. As time passes, the environment in which the software system is embedded will inevitably evolve, often in unexpected directions; the environment of a program—including its users—thus becomes input to a feedback loop that drives further evolution. A program might, at some point, perfectly satisfy the requirements of its users, but as its environment changes, the program will have to be explicitly adapted to continue doing so. In the words of Lehman, 'Evolution is an essential property of real-world software' and 'As your needs change, your criteria for satisfaction change'.

Lehman observed that successfully evolving an existing software system was a surprisingly difficult task. He summarized his observations about E-type software systems in what we today call *Lehman's Laws of Software Evolution* (adapted from [11, 3, 5])[‡]:

[‡]Herraiz *et al*. have performed a recent in-depth analysis of Lehman's Laws and modern evidence [7].

L1) *Continuing change*—A software will become progressively less satisfying to its users over time, unless it is continually adapted to meet new needs.
L2) *Increasing complexity*—A software system will become progressively more complex over time, unless explicit work is done to reduce its complexity.
L3) *Self-regulation*—The process of software evolution is self-regulating, with close to normal distribution of the product and process artifacts that are produced.
L4) *Conservation of organizational stability*—The average effective global activity rate on an evolving software system does not change over time; that is, the amount of work that goes into each release is about the same.
L5) *Conservation of familiarity*—The amount of new content in each successive release of a software system tends to stay constant or decrease over time.
L6) *Continuing growth*—The amount of functionality in a software system will increase over time, in order to please its users.
L7) *Declining quality*—A software system will be perceived as declining in quality over time, unless its design is carefully maintained and adapted to new operational constraints.
L8) *Feedback System*—Successfully evolving a software system requires recognition that the development process is a multi-loop, multiagent, multilevel feedback system; thus, for example, as a software system ages, it tends to become increasingly difficult to change because of the complexity of both the artifacts as well as the processes involved in effecting change. This law also implicitly recognizes the role of user feedback in providing impetus for future evolution.

While L8 (feedback system) was the last to be formulated, arguably, it should have been the first to be stated, as its themes pervade all of the others. As discussed earlier, Lehman based his observations on the notion that once a real-world software system has been deployed, its continued development creates a set of complex feedback loops whose effects must be considered carefully. The feedback can come from many sources including users (who may press for new features), the application domain (if changes in tax law require changes in the way a point-of-sale system calculates taxes), the technical environment in which the system runs (if changes to the operating system require changes to how some of the functionality is implemented), and the system itself (when defects are identified and need to be fixed). Developers must be keenly aware that if their software system does not respond positively to these pressures, then over time, the system will be seen as increasingly less appealing by its user base (L1: continuing change).

L2 (increasing complexity) and L7 (declining quality) imply that the changes required to evolve the system to respond to these pressures tend to make the system more complex and lower its quality, as perceived by the various stakeholders. Additional effort will be required to manage this growth in complexity and keep it under control, and development resources will have to be explicitly allocated to achieve this.

According to L3 (self regulation), over time, any measurements of the system or its process will follow a well-defined trend, with ripples in either direction that follow a normal distribution. In a way, L4 (conservation of organizational stability), is a corollary of L3: over the life of a system, the amount of work that goes into the evolution of a system remains fixed. Likewise, L6 (continuing growth) can be seen at least in part to be a corollary of L1, because change often means adding new code. Finally, L5 (*Conservation of familiarity*) states that to properly evolve a system, the team should do so in fixed increments, or it risks losing its understanding of the system.

Lehman's Laws have sometimes been criticized for lacking a solid empirical foundation; additionally, subsequent empirical studies have found important cases where the evidence does not support some of the laws [8, 9]. Lehman and his coauthors later clarified that their use of the word 'law' should be interpreted within the domain of the social sciences, and therefore, the laws are not expected to represent precise invariant relationships of measurable observations [5].

Despite these criticisms, the laws call attention to the difficulties of maintaining a deployed software system. Lehman challenged the commonly held view that software maintenance was simply the process of fixing defects found in the field, and helped explain why maintenance was becoming so costly, risky, and time consuming, and why well-designed systems often evolved into unmanageable beasts. He also popularized the view that software systems are not maintained in the traditional mechanical sense of fixing worn out pieces [10], but rather their essential properties are adapted and reshaped to meet changing expectations.

Lehman's Laws not only help to explain the risks of the future evolution of a system, but can also be seen as prescriptive advice. We know that the environment will put evolutionary pressures on any deployed software system; if we cannot predict this pressure, we can at least prepare for it. Consequently, it becomes more important to build a system that is amenable to change than to build a system that perfectly satisfies the requirements at deployment time. And once the system is deployed, we need to worry about managing and reducing the continuously growing complexity of the system. As we have learned over and over again through the years, software design is not a task that can be done entirely up-front before coding starts; instead, it needs to be done continuously and iteratively, and in response to changing needs.

## 3. SOME OBSERVATIONS ON MODERN SOFTWARE EVOLUTION AND LEHMAN'S LAWS

As noted earlier, Lehman's observations on software evolution—that he later deemed his 'laws'—were first developed during the 1960s and 1970s based on his experiences at IBM and the development of large systems such as OS/360, and he continued to refine and add to them into the 1990s [11]. However, his direct experiences—and later, the data he examined from other industrial projects—were mostly of large, systems-oriented applications written in tightly organized teams using old school management styles, programming languages, and development tools. The software world has changed a lot since then, and in the age of agile development, cloud-based services, and powerful run-time environments with comprehensive infrastructures, it is worth considering how the technical ground has shifted and how this might affect his laws. With this in mind, we now discuss some of the major recent trends of software systems and their development, and how Lehman's Laws may need to evolve to accommodate them [6].

First, we note that some recent studies cast new doubt on the likely truth of L3, L4, and L5 (for example, [9, 12]). Software development, and open source software development in particular, seems now much more ad hoc in its approach to release planning. In earlier times, software systems were developed in-house by relatively stable sets of engineers working to regular and well-defined schedules. Creating a major release of a software system was expensive, and required significant planning and budgeting. By comparison, open source systems tend not to be driven by these kinds of time and personnel pressures, and increasingly, industrial software development is echoing the style of release planning. Releases are created when a project manager decides that it is worthwhile to do so—such as when an important feature is available—rather aiming toward some largely artificial deadline. Consequently, the regularity of releases as well as the number of development artifacts created and features implemented per release tends to vary widely even within a single system [9].

The studies that Lehman and his colleagues performed to support his arguments used the metrics of the day—such as LOC, number of files touched, and number of features implemented—to measure a system's growth and complexity. However, the evolution of the very nature of software systems themselves require that we rethink how—or if—we should perform similar measurements within a modern context. We now discuss some of these concerns.

### *3.1. The emergence of software architecture*

When a large system is being developed for the first time, devising a workable software architecture is the hardest and most important design task that must be done early on. As work proceeds and understanding of the problem space improves, internal boundaries within the system begin to emerge as developers form a communal high-level mental model of the design. As these boundaries mature, their interfaces begin to harden; if they are well conceived, these interfaces can hide much of the complexity of the major components from each other. In time, some components may become more loosely coupled from the system; for example, device drivers for the Linux kernel have become less tightly coupled with the rest of the system as various simplified internal interfaces have emerged together with a facility for loading kernel modules at run-time. Overall, this has reduced the amount of knowledge about the rest of the kernel that device driver developers must understand, and so greatly simplified the task of creating new device drivers. Of course, over time, inflexible interfaces may become a problem if they are poorly thought out, and significant effort may be required to work around their flaws; Brooks would call this adding to the accidental complexity of the system [13]. But well designed interfaces can greatly lessen the amount of knowledge that developers must understand about other parts of the system.

However, the emergence of internal interfaces and subsystem boundaries means that the complexity of a large software system is probably *not* best judged as a simple sum (or product) of the size of the components; rather, a more nuanced view is needed that takes into account the complexity of those details that must be understood to interact with the various subcomponents of a software system. For example, device drivers now comprise more than 60% of the source code in the Linux kernel source distribution; yet because drivers communicate with the rest of the system via a relatively narrow interface, it is not clear that their internal complexity has much bearing on the complexity of the rest of the system and vice versa [9]. Additionally, drivers are mostly independent of each other, yet they exist in large numbers and so inflate the naive model of the size of the kernel source. Much of Lehman's empirical models use absolute numbers (the size of the system as a whole, the number and size of changes, etc.) to measure effort, complexity, and system size; we lack smarter, more sensitive models that are better tuned to the internal design of software systems.

## *3.2. The demonolithization of software systems*

Until fairly recently, large software systems were often designed as monoliths; such a system had to implement almost all of its own functionality with relatively limited help from the underlying operating system and general purpose software libraries. The current era of software development is a very different landscape: systems are embedded within an ecosphere of peer components including libraries, frameworks, run-time environments, virtual machines, and services, which in turn may be local, mobile, distributed, and location dependent [14]. Developing such a system is less about writing source code than understanding available services, evaluating security concerns, investigating distributed performance, and specifying deployment details. Thus, much of the complexity of modern development does not show up in traditional software metrics: one must evaluate possible components and services, configure their use, and deploy their systems within an appropriate run-time environment. And so, we have some thinking to do: empirical models of the size, complexity, and development effort of software systems need to explicitly recognize that not every important factor can be measured easily.

## *3.3. Open source development and agile processes*

Modern software is developed in many ways, most of which do not resemble the old school model of Big Design Up Front with waterfall-like supporting processes. Increasingly, industrial software development has been embracing the use of open source components, often contributing significant resources and implementing core functionality for such projects. Industrial developers may also take existing open source codebases, and create specializations suited to their particular needs, which are then also released to the community. For example, the Android platform is based on the Linux operating system, and both the Chrome and Safari web browsers make use of the WebKit browser engine as a core component; this means that some developers who work at Google and Apple create a significant amount of source code that is only indirectly related to their company's own proprietary products. One must therefore ask how can we evaluate a developer's contributions to an industrial project when their primary efforts are only indirectly related to it? And how can we measure characteristics of a product whose base includes a significant amount of source code that has been adapted for use rather than developed from scratch? If we are to study topics such as project evolution and developer productivity, we need new empirical models that can account for these new approaches to software development and reuse.

## *3.4. Emergent uses of software*

Lehman's eighth law (L8: feedback system) recognized that software systems are embedded within various environments that can and do provide feedback into new development. Software systems are created by a changing development team within an evolving social and technical environment; furthermore, when systems are delivered and deployed, the user community comprises another environment whose reactions can influence future development. Lehman's implicit advice is that it is better to think of software and its development as a system in the engineering sense of the term than as a passive mathematical theorem that can be manipulated formally to achieve desired goals. Although a software system is, of course, a mathematical entity in some sense, its evolution is not simply a matter of manipulating it until it performs a set of desired functionality; rather, the realities of software development processes owe much to pressures that lie outside of the formal development artifacts.

It is a truism that the Internet has changed the world forever, acting as an enabling technology for some phenomena and as a catalyst for others. The speed and volume of the feedback loop from user to developer is much stronger than anyone could have foreseen in the 1960s. But perhaps even more surprising is the number and variety of *emergent uses* some software systems have exhibited. For example, the original design goal for virtual machine platforms such as VMware and VirtualBox was simply to be able to run software written for multiple operating systems on a single computer; however, these systems are now widely used for many other purposes, including as generic deployment platforms, for malware research, and even for reproducibility of scientific experiments that require specialized software setups. Although Lehman's eighth law recognized the feedback loop from users back to developers to, for example, feed ideas for new features, these fundamentally new uses of software systems would not have been possible without the accompanying infrastructure of the internet; this represents a fundamental shift in the way that software systems may evolve.

## 4. SUMMARY

Lehman's Laws of Software Evolution were devised over a period of many years, and continue to be influential in the study of how and why software systems change over time. Despite being conceived mostly during an era when development practices were fairly rigid and tightly planned, they continue to have meaning to us today in an era of rapid change, ad hoc development practices, and wholesale reuse and adaption of third-party software assets. However, as new models of software development emerge, we must reconsider some of the original assumptions, and seek to devise new models that will continue to hold utility as we study software creation and evolution in this new era. Because Lehman's Laws concern a design space—that of software development—perhaps it is appropriate that the laws themselves seem beholden to Lehman's first law. That is, as researchers, we must seek to continually adapt Lehman's Laws over time or they will be seen as progressively less useful. This ongoing challenge is a key part of Manny Lehman's legacy.

## REFERENCES

1. Lehman MM. An interview, conducted by William Asprey. IEEE History Center, 23 Sept. 1993. Interview #178 for the IEEE History Center.
2. Dijsktra EW. On the cruelty of really teaching computing science, EWD-1036, December 1988. EW Dijkstra Archive. Center for American History, University of Austin.
3. Lehman MM. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software* 1980; **1**:213–221.
4. Lehman MM. Laws of software evolution revisited. *Proceedings of the 5th European Workshop on Software Process Technology*, EWSPT '96. Springer-Verlag: London, UK, 1996; 108–124.
5. Cook S, Harrison R, Lehman MM, Wernick P. Evolution in software systems: foundations of the SPE classification scheme: Research Articles. *Journal of Software Maintenance and Evolution* 2006; **18**(1):1–35.
6. Godfrey MW, German DM. The past, present, and future of software evolution. *Proceedings of 2008 IEEE International Conference on Software Maintenance, Track on Frontiers of Software Maintenance*, October 2008.
7. Herraiz I, Rodriguez D, Robles G, Gonzalez-Barahona JM. The evolution of the laws of software evolution. A discussion based on a systematic literature review. *ACM Computing Surveys*, (to appear) 2014.
8. Lehman MM, Ramil JF, Perry DE. On evidence supporting the FEAST hypothesis and the laws of software evolution. *IEEE METRICS*, 1998; 84–88.

9. Godfrey MW, Tu Q. Evolution in open source software: a case study. *Proceedings of 2000 IEEE International Conference on Software Maintenance*, October 2000.
10. Parnas DL. Software aging. *Proceedings of the 16th International Conference on Software Engineering*, Sorrento, Italy, May 1994.
11. Lehman MM, Ramil JF, Wernick PD, Perry DE, Turski WM. Metrics and laws of software evolution—the nineties view. *Proceedings of the Fourth International Software Metrics Symposium*, Albuquerque, NM, November 1997.
12. Robles G, Amor JJ, Gonzalez-Barahona JM, Herraiz I. Evolution and growth in large libre software projects. *Proceedings of the Eighth International Workshop on Principles of Software Evolution*, Lisbon, Portugal, September 2005.
13. Brooks FP. No silver bullet: essence and accidents of software engineering. *IEEE Computer* 1987; **20**:10–19.
14. Bennett K, Rajlich V. Software maintenance and evolution: a roadmap. *Proceedings of the 2000 International Conference on Software Engineering, track on the Future of Software Engineering*, Limerick, Ireland, May 2000.

## AUTHORS' BIOGRAPHIES

**Michael W. Godfrey** is an associate professor in the David R. Cheriton School of Computer Science at the University of Waterloo. His research interests span many areas of empirical software engineering including software evolution, mining software repositories, reverse engineering, program comprehension, and software clone detection and analysis.

**Daniel German** is professor of Computer Science at the University of Victoria. He completed his PhD at the University of Waterloo in 2000. His work spans the areas of software evolution, open source, intellectual property and computational photography.