On the Evolution of Lehman's Laws

Michael W. Godfrey
David R. Cheriton School of Computer Science
University of Waterloo, CANADA
email: migod@uwaterloo.ca

Daniel M. German
Department of Computer Science
University of Victoria, CANADA
email: dmg@uvic.ca

***Abstract*—In this brief paper, we honour 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.**

I. 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 university.[1] He got his first job in 1941 "performing maintenance on" (i.e., repairing) civilian radios; in England at the height of the second world war, 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" [9]. 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.

Like many other pioneers of computer science, Lehman lived the computer revolution from its conception. His early work was dedicated to building some of the first computers, and by the end of his life he was witnessing the ubiquity of mobile computing. It is likely that he would have spend 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.[2] This project took him to study the development of the operating system IBM S/360 and its successor, IBM S/370.

To put it into context today: IBM S/360 was an operating system for the IBM 360, a computer with up to 8 MBytes of memory; its fastest models could not reach speeds of 0.2 MIPS.[3] Lehman discovered that programmers were becoming increasingly interested in assessing their productivity, which they measured in terms of daily SLOCs and passing unit-tests. He noticed that productivity was indeed increasing, 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" [9].

It was at this time that he developed a close friendship with 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.[4] While 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. He observed that the processes involved in developing and maintaining software formed 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 other well known computer scientists, such as Edsger Dijkstra, who had a more formal view on what a programs is. For Dijkstra, a program was a mathematical entity that should be derived, iteratively, from a formal statement what it was supposed to do. In this model, you start with a precise specification, and then implement a series of formal "step-wise" refinements, gradually making it more concrete, and ultimately executable. He preferred "formal derivation of program from spec through a series of correctness-preserving transformations" over more informal and traditional views of software development, and he championed the view that teaching programming should

[1] He would later attend Imperial College London, where he received a PhD in 1957.

[2] According to Laszlo Belady, "Lehman at the time 'was on the shelf' [...] IBM never fired anyone. Instead, they put them 'on the shelf." [1]

[3] The project manager was Fred Brooks, later to become a professor and win the Turing Award.

[4] He was also instrumental in creating, at Imperial College, one the first programs in software engineering.

emphasize creating a specification and then progressively transforming it into a program satisfies it [5].[5] Lehman recognized the value of Dijkstra's position, but at the same time felt that it was not a practical model for the problem space of industrial software or for the style of development he had observed at IBM. So he postulated that programs could be divided into two main categories: S-type programs, which are derived from a rigorous specification that is stated up-front, and can be proven correct if required; and E-type (evolutionary) programs, which are strongly affected by their environment and must be adaptable to changing needs [8]. 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.[6]

II. Lehman's Laws of Evolution

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, it will have to be 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 changes". Over time requirements will change, and software must evolve to continue to satisfy these new requirements. If the environment is the one that drives the evolution, programmers are the ones who evolve the program. Lehman noted that evolving a software system was not an easy task. He summarized his observations in what we today call *Lehman's Laws of Software Evolution* (adapted from [10], [4]):

1) *Continuing change* — An E-type software system[7] that is used must be continually adapted, else it becomes progressively less satisfactory.
2) *Increasing complexity* — As an E-type software system evolves, its complexity tends to increase unless work is done to maintain or reduce it.
3) *Self-regulation* — The E-type software system's evolution process is self regulating with close to normal distribution of measures of product and process attributes.
4) *Conservation of organizational stability* — The average effective global activity rate on an evolving E-type software system is invariant over the product lifetime.
5) *Conservation of familiarity* — During the active life of an evolving E-type software system, the content of successive releases is statistically invariant.
6) *Continuing growth* — Functional content of an E-type software system must be continually increased to maintain user satisfaction over its lifetime.
7) *Declining quality* — An E-type software system will be perceived as of declining quality unless rigorously maintained and adapted to a changing operational environment.
8) *Feedback System* — E-type software systems constitute multi-loop, multi-level feedback systems and must be treated as such to be successfully modified or improved.

While Law *8 Feedback System* was the last to be formulated, arguably it should have been the first to be stated, as its themes pervade the others. As discussed above, Lehman based his observations on the notion that real world software, once deployed, forms a feedback loop. The feedback comes from many different sources, including the stakeholders (e.g., they might want new features), the environment in which the system runs (technical — e.g., new versions of the operating system might render the software system unusable unless it is adapted — and non-technical — e.g., a system for tax management might need to be updated to changes in the taxation laws), and the system itself (e.g., its own defects might need to be fixed). This feedback loop puts pressure on the system to be adapted (evolved) or it will cease to be useful to its stakeholders (Law *1 Continuing Change*).

Laws *2 Increasing Complexity* and *7 Declining Quality* imply that the changes required to evolve the system to respond to these pressures will make the system more complex and lower its quality (from the point of view of its stakeholders). Extra work will be required to manage this growth in complexity and keep it under control; otherwise, the system will eventually collapse under its own weight and cease to be useful.

According to Law *3 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, Law *4 Conservation of Organization Stability*, is a corollary of Law 3: over the life of a system, the amount of work that goes into the evolution of a system remains fixed. Finally, *Law 5 Conservation of Familiarity* states that to properly evolve a system, the team should do it 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 [11], [7]. Lehman and his co-authors later clarified that their use of the word "law"

[5]There are symmetries between their positions and those of the disciplines of engineering and mathematics; this is reinforced by the fact that Lehman viewed software as a feedback system — a typical engineering model — while Dijkstra viewed it as a mathematical concept.

[6]Originally Lehman created a third category: P-type. P-type programs cannot be specified and their development is iterative. Later, he decided that P-type programs were really a subset of E-type, and he reduced his classification to E-type and S-type programs only.

[7]Lehman used the term program, but we decided to update the descriptions to the more current term "software system".

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 [4].

Despite these criticisms, the Laws called attention to the difficulties of maintaining a deployed system. Lehman changed 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. He also popularized the view that software systems are not to be maintained in the traditional mechanical sense of fixing worn out pieces, but rather are evolved to change their essential characteristics to meet changing expectations.

The laws not only help explain the risks of the future evolution of a system, but can also be seen as prescriptive. The environment will put evolutionary pressures on any deployed software system; if we cannot predict this pressure, we can at least be prepared for it. It becomes more important to build a system that is prepared for 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. The consequence is that software design is not a task that can be done entirely up-front, before coding starts; instead, it needs to be done in a continuously and iteratively, and in response to changing needs.

III. Some observations on modern software evolution and Lehman's Laws

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 [12]. 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].

A. 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 essential 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 [3]. 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 [7]. 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.

B. The de-monolithization 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 [2]. 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.

C. 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 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 then 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 models of software development and reuse.

D. Emergent uses of software

Lehman's eighth law 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. While 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. While 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.

IV. 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. Since 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] L. A. Belady. Oral History interview by Phillip L. Frana, conducted 21 Nov. 2002, Austin Texas, 2002. OH 352, Charles Babbage Institute, University of Minnesota.

[2] K. Bennett and V. Rajlich. Software maintenance and evolution: A roadmap. In *Proc. of the 2000 Intl. Conference on Software Engineering, track on The Future of Software Engineering*, Limerick, Ireland, May 2000.

[3] F. P. Brooks. No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer*, 20:10–19, 1987.

[4] S. Cook, R. Harrison, M. M. Lehman, and P. Wernick. Evolution in software systems: foundations of the SPE classification scheme: Research Articles. *J. Softw. Maint. Evol.*, 18(1):1–35, Jan. 2006.

[5] E. W. Dijsktra. On the Cruelty of Really Teaching Computing Science, December 1988. EWD-1036. E. W. Dijkstra Archive. Center for American History, University of Austin.

[6] M. W. Godfrey and D. M. German. The past, present, and future of software evolution. In *Proc. of 2008 IEEE Intl. Conference on Software Maintenance, Track on Frontiers of Software Maintenance*, October 2008.

[7] M. W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *Proc. of 2000 IEEE Intl. Conference on Software Maintenance*, October 2000.

[8] M. M. Lehman. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, 1:213–221, 1980.

[9] M. M. Lehman. An Interview, conducted by William Asprey, IEEE History Center, 23 Sept. 1993, 1993. Interview #178 for the IEEE History Center.

[10] M. M. Lehman. Laws of software evolution revisited. In *Proceedings of the 5th European Workshop on Software Process Technology*, EWSPT '96, pages 108–124, London, UK, UK, 1996. Springer-Verlag.

[11] M. M. Lehman, J. F. Ramil, and D. E. Perry. On Evidence Supporting the FEAST Hypothesis and the Laws of Software Evolution. In *IEEE METRICS*, pages 84–88, 1998.

[12] 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*, Albuquerque, NM, November 1997.