The evolution of the laws of software evolution. A discussion based on a systematic literature review.

ISRAEL HERRAIZ, Technical University of Madrid, Spain
DANIEL RODRIGUEZ, University of Alcala, Madrid, Spain
GREGORIO ROBLES and JESUS M. GONZALEZ-BARAHONA, GSyC/Libresoft, University Rey Juan Carlos, Madrid, Spain

After more than 40 years of life, software evolution should be considered as a mature field. However, despite such a long history, many research questions still remain open, and controversial studies about the validity of the laws of software evolution are common. During the first part of these 40 years the laws themselves evolved to adapt to changes in both the research and the software industry environments. This process of adaption to new paradigms, standards, and practices stopped about 15 years ago, when the laws were revised for the last time. However, most controversial studies have been raised during this latter period. Based on a systematic and comprehensive literature review, in this paper we describe how and when the laws, and the software evolution field, evolved. We also address the current state of affairs about the validity of the laws, how they are perceived by the research community, and the developments and challenges that are likely to occur in the coming years.

Categories and Subject Descriptors: D.2.7 [**Software Engineering**]: Distribution, Maintenance and Enhancement

General Terms: Management

Additional Key Words and Phrases: Laws of Software Evolution, Software Evolution

1. INTRODUCTION

In 1969 Meir M. Lehman did an empirical study (originally confidential, later published [Lehman 1985b]) within IBM, with the idea of improving the company's programming effectiveness. The study received little attention in the company and had no impact on its development practices. This study, however, started a new and prolific field of research: *software evolution*.

Software evolution deals with the process by which programs are modified and adapted to their changing environment. The aim of Lehman's research was to formulate a scientific theory of software evolution. As any sound theory, it was meant to be based on empirical results, and aimed at finding invariant properties to be observed on entire classes of software development projects. As a result of his research some invariants were found, which were first described in [Lehman 1974] as the *laws of software evolution*.

After several years of intense activity and refinement, the last version of the laws was published in 1996 [Lehman 1996b], remaining unmodified since then. However, it is since 1996 that most studies questioning or confirming their validity have been published. We have framed our paper based on the discussions and issues raised by those studies, leading us to the following *research questions*:

Author's address: israel.herraiz@upm.es

DOI:http://dx.doi.org/10.1145/0000000.0000000

(RQ1) Why did the laws of software evolution undergo so many modifications during their early life, but have not changed in the last 15 years?
(RQ2) Which laws have been confirmed in the research literature? Which laws have been invalidated?
(RQ3) Which strategies, data and techniques can be used to validate the laws of software evolution?
(RQ4) According to the reported results, which kind of software projects, and under which conditions, fulfill the laws of software evolution?

We will address these research questions in this paper based on the literature published since the laws were first formulated. We have also explored the context of these questions, by carefully reviewing how the laws have changed over time, the reasons for those changes and the studies that have researched those changes.

The main contributions of our study can be summarized as follows:

— It is a comprehensive and systematic literature review[1] of the field of software evolution, since its inception to the most recent empirical studies, focusing on the empirical support for the laws.
— It is a detailed description of the process of progressive adaptation of the laws to a changing environment, a sort of evolution of the laws of software evolution, and how this process has been driven by the empirical results on the validity of the laws.
— It is an analysis of the current research environment and the impact of the availability of large software repositories for empirical software evolution research.
— It is a specific description of the state of the research of the laws of software evolution in the field of libre (free, open source) software development, where most cases of apparent invalidity have been reported.
— Finally we highlight new challenges and criteria for future research, based on both the original recommendations by Lehman [1974] and the availability of new research repositories.

The amount of literature we have considered is vast, ranging over 40 years. For this period, we have performed a comprehensive analysis of the most relevant publications in the field. We start with the early studies that led to the formulation of the laws, and finish with the most recent works questioning or confirming their validity.

We include not only the publications considering the laws as a whole, but also summarize and classify the empirical studies aimed at validating aspects of specific laws. In fact, this *validity question* is perhaps the expression of a crisis in the field, resulting in the laws being unrevised for more than 15 years.

Lehman's seminal book published in 1985 [Lehman and Belady 1985], which reprinted most of the studies of the sixties and seventies, is an ideal background introduction to the early period of software evolution. A more modern, general overview of the area can be found in two recent books: [Madhavji et al. 2006] and [Mens and Demeyer 2008].

The rest of this paper is organized as follows. Section 2 describes the initial formulation of the laws of software evolution. Then, the results of the systematic literature review are presented in chronological order, organized into four periods: before 1980, 1980-1989, 1990-1999 and 2000-2011. This organization into four periods fits well within the different stages in the history of software evolution as a research field, and the different versions of the laws that have been proposed. Section 3 analyses how they changed during the seventies. In Section 4 we study the different works published during the eighties, with special emphasis on the 1985 book [Lehman and Belady 1985]

[1]The details of the methodology followed for the systematic literature review are described in Appendix A.

and the first studies on the validity of the laws. Section 5 moves forward to the nineties, reviewing the works within the scope of the FEAST project. In Section 6 we describe the studies that questioned the validity of the laws in the case of libre software development. We then summarize the main findings in Section 7, providing answers to the research questions. After that, in Section 8 we propose some challenges for software evolution research in the future. In Section 9, we include a brief list of recommended readings for researchers coming to the field. Finally, Section 10 concludes this paper. The details and methodology about the systematic literature review are given in Appendix A, included at the end of the paper.

2. THE ORIGINAL LAWS

The laws of software evolution were presented for the first time in Lehman's inaugural professorship lecture at the Imperial College London [Lehman 1974] (reprinted as [Lehman 1985c]). Initially, Lehman proposed three laws, shown in Table 2, stating three basic principles for the evolution of software systems:

— Software systems must be continuously changed to adapt to the environment.
— Changes increase the complexity of software.
— Software evolution can be studied using statistical methods.

The first two laws show the fundamental idea behind software evolution. Software must change, but current changes make future ones more difficult to do, because the former increase complexity. This only happens for *large* software projects, not considering the size of the product but the size and structure of the project team. Thus, the laws of software evolution should be applied only to projects with several managerial levels.

Interestingly, if we compare the above principles with table 2, we observe that Lehman did not use the term complexity, and referred instead to *entropy*. Lehman chose this term to highlight the fact that changes introduce disorder, and therefore the structure of programs *degrades*, making systems more difficult to comprehend, manage and change. However, in the rest of the 1974 paper he used the term complexity instead.

Lehman was referring to what today would be labeled as architecture complexity of the system. He distinguished between three levels of complexity: *internal*, *intrinsic* and *external*. Internal complexity has to do with the structural attributes of the code. Intrinsic complexity is related to the dependencies between different parts of the program.[2] Finally, external complexity is a measure of the understandability of the code provided documentation for it is available.

The third law was the first proposal for a statistical approach to the study of software evolution. The literal text of the law (see Table 2) refers to stochastic growth trends, pointing out that these trends are self-regulating and statistically smooth. Because the trends are statistically smooth, they can be studied and forecasted using statistical methods. Lehman suggested the use of regression techniques, autocorrelation plots and time series analysis for the study of software evolution, with a rationale based on the idea of feedback trends. However, the research environment of the time was very constrained: access to software projects was difficult and scarce, and such statistical approaches were difficult to implement in practice. Current research environments are much richer, making this statistic approach possible (see Section 8.1).

[2]A large program has a defined task that is divided in many subtasks. The intrinsic complexity is related to the relationships and the extent of those subtasks.

Table I. Laws of software evolution in 1974 [Lehman 1974] (reprinted as chapter 7 in [Lehman 1985c]).

I	*Law of continuing change* A system that is used undergoes continuing change until it becomes more economical to replace it by a new or restructured system.
II	*Law of increasing entropy* The entropy of a system increases with time unless specific work is executed to maintain or reduce it.
III	*Law of statistically smooth growth* Growth trend measures of global system attributes may appear stochastic locally in time and space but are self-regulating and statistically smooth.

Table II. Laws of software evolution in 1978 [Lehman 1978].

I	*Law of continuing change* A program that is used undergoes continuing change or becomes progressively less useful. The change process continues until it is judged more cost effective to replace the system with a recreated version.
II	*Law of increasing complexity* As a large program is continuously changed, its complexity, which reflects deteriorating structure, increases unless work is done to maintain or reduce it.
III	*Law of statistically regular growth* Measures of global project and system attributes are cyclically self-regulating with statistically determinable trends and variances.
IV	*Law of invariant work rate* The global activity rate in a large programming project is invariant.
V	*Law of incremented growth limit* For reliable, planned evolution, a large program undergoing change must be made available for regular user execution (released) at maximum intervals determined by its net growth. That is, the system develops a characteristic average increment of safe growth which, if exceeded, causes quality and usage problems, with time and cost over-runs.

3. THE LAWS IN THE SEVENTIES

The first version of the laws of software evolution was based on a single case study, the OS/360 operating system [Belady and Lehman 1976]. Soon, Lehman started to search for further empirical support [Lehman and Parr 1976], following the statistical approach introduced in his 1974 lecture [Lehman 1974]. The new results led to the modification of the original laws as well as to the addition of two new ones [Lehman 1978] (see Table 3).

Although the first three laws kept their essential meaning, their formulation changed. The first law, the basic principle of software evolution, now includes a clarification: software must change or it becomes less useful.

In the second law *entropy* was changed to *complexity*, the term used in the research context of the time. This context can be better understood thanks to the analysis by Belady of the research on software complexity during the seventies [Belady 1979] (reprinted as [Belady 1985]).

The third law was modified to include a mention of "statistically determinable trends and variances". While working at IBM, Lehman gained access to a subset of metrics regarding the evolution of OS/360: size of the system (in number of modules), number of modules added, removed and changed, release dates, amount of manpower and

machine time used, and costs involved for each release. When he plotted these variables over time, plots resulted to be apparently stochastic. However, when averaged, variables could be classified in two groups: some of them grew smoothly, while others showed some kind of conservation (either remained constant, or with a repeating sequence). Lehman started to think of software evolution as a *feedback driven* process, self-regulated, whose properties (trends, variances) could be estimated by empirical studies based on statistical methods. He did not explicitly mention the term *feedback* at this time, but it became a fundamental concept of the software evolution field during the nineties.

The fourth law states that the work rate remains invariant over the lifetime of the project. In other words, using the terms of the third law: there is a statistically invariant trend in the activity rate of the project. That invariance is probably due to the feedback process governing the evolution of the system. This law can be understood as a subcase or corollary of Brooks' law [Brooks 1978]: regardless of the variations in manpower in a software project, work rate remains invariant.

The fifth law states that there is a *safe* growth rate, and that the interval between releases should be kept as wide as possible to maintain the growth rate under control. This reflects the release practices of the time, when software was shipped through mail and courier, and the distribution costs were much higher than today. Under those constraints, it is better to make sure that products are released only when they are ready, rather than releasing early and continuously send updates to users (an extended practice nowadays).

4. THE LAWS IN THE EIGHTIES

The eighties saw the birth of the seminal book on software evolution [Lehman and Belady 1985], still called *Program Evolution* at the time. It reprinted many of the original works in the field, which maybe would have been lost otherwise.[3] It contains not only the laws of software evolution as formulated at the time, but also all previous works, which makes it an invaluable resource to reconstruct the history of software evolution. It helps to understand the environment where the laws of software evolution were conceived. Finally, it also helps to realize how the laws are not immutable, and that they have undergone significant change since their original conception.

The book reprinted the first mathematical model of software evolution [Woodside 1985], which was proposed by Woodside [1980]. This was the first attempt to simulate the evolution of software using the concepts behind the laws of software evolution. Woodside's model was based on a balance between *progressive* and *anti-regressive* work in the software process. As Lehman had shown in the previous decade, progressive work introduces new features in the system, while anti-regressive work attempts to maintain the program well-structured, documented and evolvable. This balance between *progressive* and *anti-regressive* work will be the base of software evolution simulation models in the next decades, which will be used to validate the laws, as we will show in Section 5.2.

This decade also saw the stabilization of the evolution framework, with new and more elaborated concepts such as the SPE scheme (described in Subsection 4.1), that refined the notion of *large* programs of the seventies.

The second chapter of the 1985 book [Lehman 1985a] is a good summary of the state of the art of the decade, including the laws of program evolution, the SPE classification scheme, the notion of feedback and its influence on software evolution, and a discussion

[3]The book is out of print, and is kindly redistributed with permission of the author and the publisher by the ERCIM Working Group of Software Evolution at http://wiki.ercim.eu/wg/SoftwareEvolution/index.php/Publications (consulted October, 2012).

on the need for a theory of software evolution. The second chapter reprints a paper that appeared a year before [Lehman 1984].

Finally, during these years the laws faced the first studies questioning their validity, including the PhD dissertation by Pirzada [1988], who proposed some changes to the laws such as making the third law specific only to commercial software.

Let us discuss these points further in the following subsections.

4.1. The SPE scheme

In the early 1980s Lehman abandoned the notion of *large programs*, modifying the laws to adapt to a new scheme, introduced with their new formulation [Lehman 1980]. Lehman narrowed the application of the laws to large programs because the evolutionary behavior of software was different in small ones.

Large programs are usually developed by large teams with more than one managerial level, and have at their disposal a large user base which could provide feedback to the programming processes. In addition, these programs usually produce some development and maintenance logs which could be analyzed to test the laws.

However, this definition of *large* could not be applied without problems and uncertainty: there was no clear distinction, and two programs of similar size could behave differently. Because of this, the domain of applicability of the laws was changed from program size to the so-called *SPE scheme* classification. Under this scheme we can find three classes of programs:

— S-type (specified) programs are derivable from a static specification, and can be formally proven as correct or not.
— P-type (problem solving) programs attempt to solve problems that can be formulated formally, but which are not computationally affordable. Therefore the program must be based on heuristics or approximations to the theoretical problem.
— E-type (evolutionary) programs are reflections of human processes or of a part of the real world. These kind of programs try to solve an activity that somehow involves people or the real world.

The laws of software evolution were said to be referring only to E-type software. Because the world irremediably changes, this kind of software must be changed to keep it synchronized with its environment. Software requirements also change, because human processes are hard to define and state, which also lead to modifications in the software. Therefore it is likely that a program, once implemented, released and installed, will still need further changes requested by its users. Also, the very introduction of the system in the world will cause further demands for changes and new features. This last source of changes causes a feedback loop in the evolution of the program.

Unlike E-type, S-type software does not show an evolutionary behavior. Once the program is written, it is either correct or not with respect to a specification. If it is not correct, it will not be released, and there is no chance for evolution (by definition, it happens after the release of the program). However, if the program is correct, it is finished and there is no chance for further refinement or adaptation. This means that it will not reach the evolution stage either. In summary, because the specification of this type of programs is static, the program will remain the same regardless of any change in the environment.

The case of P-type software falls between the E and S-type software. For P-type software the specification cannot be completely defined before the implementation of the software. Thus, after releasing the system the specification may still be subject to change, and it follows P-type software evolution. However, the nature of the specifications of P-type software is different to that of E-type software. The problem to be specified does not change, but only our understanding of it. In E-type software, the

Table III. Laws III, IV and V were changed in 1980 [Lehman 1980; 1979], and remained unchanged in the book published in 1985 [Lehman and Belady 1985].

III	*The Fundamental Law of Program Evolution* Program evolution is subject to a dynamics which makes the programming process, and hence measures of global project and system attributes, self-regulating with statistically determinable trends and variances.
IV	*Conservation of Organizational Stability (Invariant Work Rate)* During the active life of a program the global activity rate in the associated programming project is statistically invariant.
V	*Conservation of Familiarity (Perceived Complexity)* During the active life of a program the release content (changes, additions, deletions) of the successive releases of an evolving program is statically invariant.

environment around the system changes. Therefore, the evolution of *P*-type software is different to that of *E*-type. Lehman did not pay special attention to *P*-type software, and focused on explaining the evolution of *E*-type software.

4.2. Changes to the laws in the 80s

In the early eighties, Lehman published a new version of Law III, IV and V [Lehman 1979; 1980], as shown in Table 4.2. The name of the third law was changed to *Fundamental Law of Program Evolution*, highlighting again the fact that software evolution is "self-regulating with statistically determinable trends and variances". Lehman also included a mention of the measures of process and product metrics that can be determined using statistical methods. In essence, the law is the same as in previous formulations. The change in the name reflects the importance that Lehman gave to this law, converting it in an essential property of evolution, and hence into a cornerstone of a theory of software evolution.

The fourth law remains the same in its essence too, referring to the stability within the organization that develops and maintains the system. Sudden and substantial changes are not possible in an organization that develops an *E*-type system. This causes a conservation of work rate in the programming project. In previous formulations, the law explicitly mentioned *large projects*, which are the kind of projects usually developed by organizations with several managerial levels, e.g., stable organizations. The term *large project* was substituted by the "associated programming project", thus discarding the notion of large programs and adapting the law to the SPE scheme, introduced soon after this law was formulated.[4]

The fifth law had major changes made to it. The *safe growth rate* notion of the seventies was discarded, and the law name was changed to "conservation of familiarity". In summary, it states that the activity rate of a project remains constant during its active life, and is closely related to the fourth law (constant work rate). Lehman formulated this law after analyzing the amount of change between releases. If a release varied widely from a previous release, it was followed by releases with less changes, making the average amount of change constant over a large number of releases. The amount of change is also related to the familiarity of the stakeholders (not only users) with the system. With new releases, stakeholders need some effort to learn how behavior of the system has changed. They need to recover their familiarity with the system, and for that they have to invest some time. Once they are familiar again with the system, the

[4]The discussed modification in the law appeared for the the first time towards the end of 1979 [Lehman 1979], some months before the SPE scheme was published [Lehman 1980].

perceived complexity of the system will again be close to that of the previous release. Thus the average perceived complexity over all the releases should be constant. There is no mention to the influence of the growth rate on the quality of the system, suggesting that Lehman decided to drop the idea of a *safe* release rate. A similar idea of a constrained growth rate was re-introduced in the 90s, as will be shown in Section 5.1.

4.3. First validating research

One of the first works to address the issue of the validity of the laws of software evolution using a comprehensive statistical approach was the PhD thesis by Pirzada [1988]. In his thesis, he studied the evolution of several *flavors* of UNIX. At the time of the study, the evolvability and maintainability of those UNIX versions was a subject of concern. Applying the approach suggested by Lehman and Belady [1985], he did not only study the evolution of UNIX, but evaluated the validity of the predictions of the laws as well.

The versions of UNIX under study were divided into three branches (or *streams*, in the terminology used by Pirzada): research, academic, and supported and commercial. The versions of UNIX in each *stream* were the following:

— **Research stream**
The original UNIX version created by Ken Thompson and Dennis Ritchie at Bell Labs. Pirzada studied nine versions of this flavor of UNIX, released between 1971 and 1987.
— **Academic stream**
Versions of UNIX developed by the Computer Systems Research Group (CSRG) at the University of California at Berkeley, the Berkeley Software Distribution (BSD). The last release considered by Pirzada, 4.3 BSD, appeared in 1986.
— **Supported and commercial stream**
Versions developed by the UNIX Support Group (USG), a small team created by the Switching Control Center System at Bell Systems, between 1983 and 1986.

The conclusions arrived at by Pirzada were: all the streams verified the first law (continuing growth), but the situation was different for the others. Only the supported and commercial streams evolved according to the laws of software evolution. This stream showed a slowdown pattern, because of its increasing complexity. This is the typical pattern predicted by the laws of software evolution: if we measure size over time, the growth rate decreases over time. More interestingly, the academic and research streams were growing rapidly, and the academic stream was even accelerating (the growth rate increased over time). These growth patterns are not compatible with the laws of software evolution.

According to Pirzada, processes in pure commercial environments were more constrained, and therefore commercial software was much more likely to exhibit structural deterioration (second law). Only software developed in pure commercial environments evolved according to the formulation of the laws. This has been also verified more recently [Siebel et al. 2003], with a system that was developed in an academic environment, and was subsequently adapted to a commercial environment. The diversity of the processes and the change in the evolution of the system affect the quality of products and processes.

The second law of software evolution (as formulated at the time, see Table 3) states that continuous changes cause the structural deterioration of the system. This deterioration increases the complexity of future changes, and the net result is a pattern of decreasing growth. Pirzada found evidence conforming to this law only for the commercial stream, but the increase in complexity is validated only *after* the commercializa-

tion of the products. Pirzada concluded that commercial pressures enforce constraints to the growth of software and foster the deterioration of the system.

Law by law, supported and commercial UNIX conformed with the third, fourth and fifth laws (as formulated at the time, see Section 4.2). The third law could not be validated for the academic and research streams.

Pirzada cited some earlier works highlighting the controversy regarding the universality of the laws. From all those works [Lawrence 1982; Chong Hok Yuen 1980; Benyon-Tinker 1979; Kitchenham 1982], we could only gain access to a paper by Lawrence [1982], which concluded that the third, fourth and fifth laws were not confirmed in different case studies. However, Lawrence found that the first and second laws were validated.

5. THE LAWS IN THE NINETIES

In 1989, the *Journal of Software: Evolution and Process*[5] published its first issue, which included an article by Lehman about the "Principle of Uncertainty" in software technology [Lehman 1989]. This principle of software uncertainty was already scattered across the text of the laws of software evolution in previous publications. Lehman distilled and condensed the principle, which became another essential property of software evolution. The principle, in Lehman's own words [Aspray 1993], states:

> No *E*-type program can ever be relied upon to be correct.

In other words, *E*-type software is never finished, and keeps evolving in order to (i) fix defects introduced in the previous programming activities, and (ii) take into account new demands from users.

In the case of the theory of software evolution, the principle of software uncertainty refers not only to defects, but to change requests in general, which can be also about missing or desired functionality. Because *E*-type software is a model of the world, and the world is continuously changing, the only fate of software is to change or die. This is the main driving force explaining the laws of software evolution. In Lehman's opinion, the mission of Software Engineering is precisely to mitigate uncertainty through the introduction and control of processes, methods and tools.

The principle of uncertainty was further developed during the following years [Lehman 1990; 1991], and finally included as a part of a broader theory of software evolution in the following decade [Lehman and Ramil 2002b].

Software uncertainty led Lehman to develop additional laws of software evolution. In a footnote in one of his papers published during this decade [Lehman 1991], he discusses the relationship of uncertainty with the notion of *domain*. Lehman mentions the need for a new sixth law that would reflect the inevitability of software evolution because the scope of the system will irremediably increase over time. The increase of the scope is due to increasing user demands, which is, in other words, the formulation of the principle of software uncertainty. However, this law was not formulated in that publication. It appeared several years later, as we will discuss in the next subsection.

5.1. Dimensions of software evolution: adapting and expanding the laws to a changing environment

In 1994, the invited keynote of the International Conference on Software Maintenance was entitled *Dimensions of Software Evolution* [Perry 1994] (republished as [Perry

[5]The name between 1989 and 2000 was *Journal of Software Maintenance: Research and Practice*, and between 2000 and 2011 it was *Journal of Software Maintenance and Evolution Research and Practice*. In 2011 it was merged with the journal *Software Process: Improvement and Practice*, and the name changed to its current form.

2006]). In his keynote, Perry argued that software evolution depended not only on the age, size or stage of a project, but also on the nature of its environment. This fact is interesting, because all the research on the validity of the laws had focused on the nature of the environment where each study took place (e.g. the different streams used by Pirzada [1988]).

These works led to the reformulation of the laws of software evolution in 1996 [Lehman 1996b]. Lehman added three new laws and adapted the rest, as shown in Table 5.1. This was the last reformulation of the laws.

Lehman *et al.* [1997] found empirical support for all the reformulated laws, but the fifth (conservation of familiarity) and the seventh (declining quality). Some other studies found similar empirical support as well [Gall et al. 1997].

The changes in the laws were caused by results published in previous decades, rather than by studies performed during those years. One of the main changes consisted in the inclusion of the term "*E*-type", to highlight that the laws are only valid for that kind of software. The **third law** changed its name again, now to *Law of Self Regulation*. It was also briefer, just stating that software evolution is feedback-regulated. There is an interesting change in the **fifth law**, *conservation of familiarity*, now highlighting that the growth rate is constrained by the need to maintain familiarity. This is similar to the *safe growth rate* of this law during the seventies: any project has a limit in how fast it can grow without suffering of problems due to the growth rate.

The **sixth law**, derived from the *Principle of Software Uncertainty*, was called "*of continuing growth*". It was already mentioned in 1991 [Lehman 1991], although it is first formulated in 1996. It states that not only software will undergo changes, but that some of those changes will be performed to increase the functionality of the program. Lehman attributed this to the need of increasing functionality due to missing requirements in the original specification of the system, and to feedback-generated demand, rather than to defects.

The **seventh law**, of *declining quality*, states that unless work is carried out to avoid it, quality (and hence user satisfaction) will decline as time passes. We can consider the seventh law a corollary of the others, as the decline of quality is due to the degradation of the system under continuous changes.

The **eighth law** was derived by Lehman after observing the organizations that produced the software which he used as case studies for his empirical work. These projects often involved several managerial levels: they were multi-level, multi-loop and multi-agent feedback systems. This, and the apparent conservation of some quantities (like the fourth and fifth laws show), led to the formulation.

Feedback in the software evolution process became the cornerstone of Lehman's work during the 90s, with the *Feedback, Evolution and Software Technology* (FEAST) research project, focused on the so-called *FEAST hypothesis* [Lehman 1996a]. This hypothesis states that software evolution is a complex feedback learning system, which is hard to plan, control and improve.

5.2. The FEAST project

The software evolution field was focused on the the validity of the laws during the nineties, with the FEAST project being one of the main sources of empirical works during that time. This effort resulted in the validation of the laws both by simulation [Chatters et al. 2000; Wernick and Lehman 1999] and by empirical methods [Lehman et al. 1997]. In the scope of the FEAST project, Turski produced the *reference model for smooth growth of software systems* [Turski 1996], which was generalized some years later [Turski 2002]. The FEAST project leaded to the recognition of software evolution as a subject of research on its own. It also started to systematize and formalize the field.

Table IV. Laws of software evolution in the nineties, published in 1996 [Lehman 1996b] and republished in 2006 [Lehman and Fernández-Ramil 2006]. This can be considered the current formulation of the laws of software evolution.

I	*Law of Continuing Change* An *E*-type system must be continually adapted, else it becomes progressively less satisfactory in use
II	*Law of Increasing Complexity* As an *E*-type is changed its complexity increases and becomes more difficult to evolve unless work is done to maintain or reduce the complexity.
III	*Law of Self Regulation* Global *E*-type system evolution is feedback regulated.
IV	*Law of Conservation of Organizational Stability* The work rate of an organization evolving an *E*-type software system tends to be constant over the operational lifetime of that system or phases of that lifetime.
V	*Law of Conservation of Familiarity* In general, the incremental growth (growth rate trend) of *E*-type systems is constrained by the need to maintain familiarity.
VI	*Law of Continuing Growth* The functional capability of *E*-type systems must be continually enhanced to maintain user satisfaction over system lifetime.
VII	*Law of Declining Quality* Unless rigorously adapted and evolved to take into account changes in the operational environment, the quality of an *E*-type system will appear to be declining.
VIII	*Law of Feedback System* *E*-type evolution processes are multi-level, multi-loop, multi-agent feedback systems.

The nineties also saw the birth of the *International Workshop on Principles of Software Evolution*. In the first edition Lehman and Wernick [1998] presented a simulation model of the impact of feedback on software evolution with some of the results obtained in the FEAST project.

Simulation was in fact one of the main techniques used in FEAST to validate the laws, finding confirmation for seven of them. Only the second law, of increasing complexity, could not be confirmed due to the lack of empirical data [Lehman and Ramil 1999]. The project also found empirical evidence confirming the same seven laws, showing that Turski's model [Turski 1996] could be accurately fitted to empirical data extracted from industrial projects [Lehman et al. 1998b].

The results of the FEAST project were summarized in a paper by Lehman and Ramil that appeared in 2002 [Lehman and Ramil 2002a]. Lehman also published another good summary of the field during the nineties, with a focus on the implications for the software industry [Lehman 1998].

But besides those carried out at FEAST, the field still lacked further empirical studies [Kemerer and Slaughter 1999]. Above all, more consistent studies were needed that could allow for the meta-analysis of results [Bennett et al. 1999].

6. SOFTWARE EVOLUTION IN THE 2000S

In the decade of the 2000s the field reached its maturity with the publication of two books [Madhavji et al. 2006; Mens and Demeyer 2008].

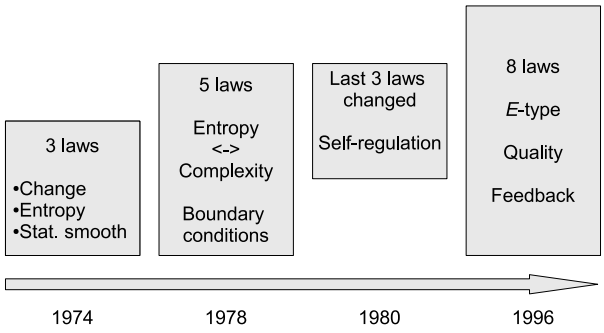

Fig. 1. Diagram of the evolution of the laws of software evolution.

Since its inception in the late sixties and early seventies of the previous century, the field of software evolution kept evolving, adapting the laws or stating new ones (see Figure 1). But the laws of software evolution have remained invariant since 1996. The only publication written by Lehman that explicitly formulates the laws in the last decade is a chapter [Lehman and Fernández-Ramil 2006] in the book published in 2006 [Madhavji et al. 2006].[6] That book included a slight clarification of the SPE scheme [Cook et al. 2006a] (also published as [Cook et al. 2006b]), but the modification did not impact the formulation of the laws.

This decade also saw the publication of some of the main results of the FEAST project. One of them was a quantitative model for the simulation of the software process [Ramil et al. 2000]. Simulation could be applied to manage long-term software evolution and control complexity in the software process [Kahen et al. 2001; Lehman et al. 2002]. These results led to a paper proposing a roadmap for a theory of software evolution [Lehman and Ramil 2001a].

The FEAST project also produced a guide to control, plan and manage the software process [Lehman and Ramil 2001b]. The guide addressed some of the criticisms about the lack of formalization and absence of precise definitions of the laws of software evolution.

In fact, this lack of formalization and precise definitions are two of the causes that explains the proliferation of studies about the validity of the laws. The FEAST project mentioned some of these validity studies, with conflicting results: first, the case of the OS 360/370 defense system (mentioned in [Lehman and Ramil 2002a]), and second, the Linux kernel, by Godfrey and Tu [2000]. The latter, which found some deviations from the laws, was labeled as an *anomaly* by Lehman *et al.* [2001].

The availability of data thanks to the open nature of libre software development, and the attention given by Lehman to the case of Linux and to the question of the validity of the laws, fostered a number of empirical studies of libre software evolution during this decade, leading to a growth by an order of magnitude in the number of works in the field (see Figure 2 and Section A.3 in the Appendix).

The popularity of the studies of the validity of the laws in libre software finally allowed for meta-studies comparing results for different projects and with different research methodologies [Fernández-Ramil et al. 2008].

[6]In that book, and in a chapter in another software evolution book [Mens and Demeyer 2008], Ramil signs his works as Juan Fernández-Ramil, while in all his earlier works he appears as Juan F. Ramil. Although obviously both names correspond to the same author, we have cited his works using the name as it appears in each paper.

6.1. Are the laws valid for libre software?

The first studies trying to verify the laws of software evolution for libre software were based on a relatively small set of case studies. Initially, Godfrey and Tu studied only the case of the Linux kernel [Godfrey and Tu 2000; 2001], finding that it was evolving at an accelerating pace, contrary to the predictions of the laws. These results were confirmed by Robles *et al.* five years later [Robles et al. 2005], extending the results to a set of 19 projects.

More recently, Israeli and Feitelson [2010] also studied the Linux kernel, trying to find out whether it fulfilled the laws or not. They found that the superlinear growth pattern that had been found in 2000 by Godfrey and Tu, and confirmed in 2005 by Robles *et al.*, stopped with the release 2.5 of Linux. From that release on, growth has been linear.

This change in the growth pattern coincides in time with the changes in the release policies of Linux. Up to the 2.5 release, Linux development occurred in two parallel branches: stable and unstable. New functionality was added to the unstable branch, that eventually became the stable branch. The stable branch only accepted bug fixes, and its growth pattern was completely flat (i.e., zero growth). However, with the 2.6 release, the procedure changed, and both stable and unstable development occurs in the same branch.

Contrary to the previous studies, Israeli and Feitelson concluded that Linux confirmed most of the laws. In particular, it fulfills those related to the growth and the stability of the process. The laws of increasing complexity, conservation of familiarity and declining quality were contradicted though:

— The second law (increasing complexity)
  The average cyclomatic complexity is decreasing in Linux. This is due to the high rate of aggregation of small functions, that dilute the average value of the complexity.
— The fifth law (conservation of familiarity)
  This law is only validated for minor releases. Major releases lead to a sudden change in the familiarity with the system, because they introduce severe changes.
— The seventh law (declining quality)
  The quality of the system is increasing over time, as measured by the maintainability index.

Moreover, some of the laws are only partially validated:

— The third law (self-regulation) was validated using indirect empirical support, analyzing the shape of the incremental change in files over time. This plot oscillates around an average value, in what seems to be a signal of self-regulation.
— The fifth law (conservation of familiarity) is only valid for minor releases. From time to time, Linux creates major releases with substantial changes. This leads to discontinuous familiarity: it is only conserved through the minor releases between major releases.
— The eighth law (feedback system) was not empirically validated, but it was assumed to be true considering how the Linux project works, i.e., driven by a community of users and developers. However, Linus Torvalds, the leader of the project, has a strong influence in the project, so it is not clear whether the project is completely feedback driven.

The main conclusion of Israeli and Feitelson's study is that the *perpetual development* model[7] of libre software is compatible with the laws of software evolution. In a

[7]The *perpetual development* model is a lifecycle model for *E*-type software where most of the changes occur in a continuous manner. When new features accumulate, the project is frozen to prepare for a new major

typical textbook software lifecycle model, there is a clear division in two main stages: development until first release, and maintenance thereafter. This is not the usual pattern found in Linux, which evolves continuously with frequent releases. The system is developed in collaboration with users, with the community driving the evolution of the project. This model fits the feedback-driven paradigm, and so authors conclude that the perpetual development model is a good description of the lifecycle of *E*-type software.

Koch [2005; 2007] addressed the validity question with a large sample of software projects. He extracted a sample of $8,621$ projects from `SourceForge.net`, a hosting site for libre software projects. His methodology is similar to that found in the works of Godfrey and Tu, and Robles *et al*. Koch measured the size in lines of code for all the projects, and fitted a regression model to each curve. He found most of the projects to grow either linearly or with a decreasing rate, as predicted by the laws. However, around 40% of the projects showed a superlinear pattern, incompatible with the laws. Those projects were usually large, in terms of code size, with a high number of participants, and a high inequality in the number of contributions [Mockus et al. 2002]. Koch speculated that the cause of the superlinear growth might be a certain organizational model, that he calls the *chief programmer team*, a term originated in IBM in the seventies [Baker 1972]. This organizational model seems to affect the growth rate, allowing for these patterns of increasing growth.

Therefore, at a first glance, the studies by Koch showed that a majority of projects evolve according to the behavior predicted by the laws. However, `SourceForge.net` is known to contain many small and pet projects, many of which are abandoned [Rainer and Gale 2005]. That means that, for the kind of regression analysis used by Koch, projects that stopped evolving would appear as conforming to the laws of software evolution. But non-evolving projects cannot be considered *E*-type software. In addition, Koch reports that 67.5% of the sample are projects with only one developer, which would make them fall under the category of pet projects. If we focus on projects that fit the perpetual development model described in Israeli and Feitelson [2010], we can only consider the results obtained for large projects, with several developers and a large user base. That implies that the consequences of Koch's study are that most of the permanent-evolving projects grow with accelerating paces, without constraints, which is contrary to the predictions of the laws of software evolution.

A more recent study explores different effort models for libre software development, comparing them with effort models used traditionally in closed-source software [Fernández-Ramil et al. 2009]. One of the conclusions of the study is that complexity does not slow down the growth of the analyzed libre software projects, contrarily to what is stated by the laws. The authors suggest that libre software is *more effective* than closed-source, and therefore less effort is required to develop projects of similar sizes.

In 2008, Fernández-Ramil *et al.* [2008] reviewed and summarized the main empirical studies about the evolution of libre software, and how they helped to confirm (or not) the laws of software evolution. In their opinion, libre software development follows a process more chaotic than in-house development, which may be an explanation of the divergence.

Some libre software projects evidence discontinuities in their evolution. Similar patterns were found as well in the nineties in the FEAST project, being this the reason why the fourth law included the clarification "or phases of that lifetime" (see Table 5.1).

---

release. After its publication, development continues in the project towards the next major release. However, bug fixes are accepted are still accepted for the past releases, which are distributed in the form of new minor releases.

A software project goes through different stages during its lifetime. There are different models for the stages of a software project. The simplest one includes two stages: development and maintenance. Other models divide the software lifetime in five stages [Rajlich and Bennett 2000]. In any case, studies reviewed in [Fernández-Ramil et al. 2008] did not take into account this possibility. Discontinuity means that growth rates may suddenly change when there is a phase transition, and therefore growth rates measured with low level parameters (like lines of code) can not be trusted to judge the validity of the laws. Interestingly, the complete study about Linux by Israeli and Feitelson [2010] found that the growth of Linux changed from superlinear to linear when the project entered a new phase, i.e., the change from having unstable and stable branches in parallel to a single development branch.

Another source of controversy in the validation of the laws has been their lack of formalization. According to Fernández-Ramil *et al.*, each law can be formalized in more than one way. In the original empirical works that led to the formulation of the laws, the studied aspects and properties were of a high-level nature, i.e., they were defined at a high-level of abstraction. However, the empirical studies about libre software use lower level parameters, which suffer more variability and are harder to predict.

After analyzing the common and diverging points of the reviewed papers (some of them already mentioned in this paper [Godfrey and Tu 2000; Robles et al. 2005; Herraiz et al. 2006]), Fernández-Ramil *et al.* [2008] summarized the state of the validation question, for each one of the laws:

— Continuing change (I)
  This law applies well to libre software, because successful projects are continuously developed over time. However, in some occasions even successful projects can experience periods without activity.
— Continuing growth (VI)
  Although some projects may experience periods with a flat growth, successful libre software projects continuously increase their functionality, and therefore grow continuously.
— Declining quality (VII)
  In this case, the confirmation is difficult to test in practice, because it depends on how quality is measured. However, the number of defects found in libre software projects tends to grow, meaning that quality might decline over time, and therefore this law is possibly confirmed.
— Feedback system (VIII)
  Libre software projects are feedback-driven, as predicted by the laws, but the feedback process is more chaotic than in the case of in-house software development. This is probably because libre software projects are open systems. This means the development team is far from constant over time, with contributors joining and leaving, with code being sometimes duplicated, or imported from other projects.

The rest of laws could not be confirmed with the studies included in their review [Fernández-Ramil et al. 2008]. However, this does not mean that the laws have been invalidated, and the authors recommend further work in the matter.

6.2. Status of the validity question for libre software

Table 6.2 shows some selected references that have attempted to validate the laws of software evolution for the case of libre software.

Bauer and Pizka [2003] confirmed the validity of all the laws using a sample of selected libre software projects. Their review is qualitative: they focus on the software process of the selected projects, on their high level architecture, and on the involvement

Table V. Main studies about the validity of the laws for libre software. The laws marked with "(p.)" are only partially validated.

Ref.	#	Approach	Validated laws	Invalidated laws
[Godfrey and Tu 2000]	1 (Linux)	Size metrics (overall, submodules). Growth rate.	I, VI	II, III, IV, V
[Godfrey and Tu 2001]	4	Size metrics (overall, submodules). Growth rate.	I, VI	II, III, IV, V
[Bauer and Pizka 2003]	8	Qualitative review of some selected projects	I-VII (all laws)	None
[Robles et al. 2005]	19	Size metrics (overall, submodules). Growth rate.	I, VI	II, III, IV, V
[Herraiz et al. 2006]	13	Size metrics (only overall, different metrics, quadratic regression model). Growth rate.	I, VI	II, III, IV, V
[Koch 2005] & [Koch 2007]	8621	Size metrics (overall level, quadratic regression model). Growth rate.	I, VI	II, IV (p.).
[Israeli and Feitelson 2010]	1 (Linux)	Deep analysis (submodules). Size, complexity, quality metrics	I, III (p.), IV, V (p.), VI, VIII (p.)	II, V (p.), VII
[Neamtiu et al. 2013]	40	Size and defects density Overall level.	I, VI	II, III, IV, VII, VIII
[Vasa 2010]	40	Size, object-oriented and complexity metrics. Overall level.	I, III, V, VI	II, IV, VII

of companies in the projects. They conclude that all the laws are verified in the all case studies. However, there is no further empirical evaluation.

The next study [Herraiz et al. 2006] is similar to the study by Robles *et al.* [2005], although it only measured size at the overall level, using different metrics. The same regression approach was used, but this time also fitting a quadratic model. If the quadratic correlation coefficient was positive, the growth was assumed to be superlinear, sublinear if it was negative, and linear if it was close to zero. Some cases presented a superlinear pattern, as in the three previous studies. Some other were either linear or sublinear. Therefore, the invalidation or validation of the second and fourth laws were only partial.

In a study of 705 releases of 9 open source software projects, Neamtiu et al. [2013] reported that only the laws of continuing change and continuing growth are confirmed for all programs.

Vasa [2010] studied 40 large open source systems finding support for laws I (Continuing Change), III (Self Regulation), V (Conservation of Familiarity), and VI (Continuing Growth). His methodology was based on size measurements, and object-oriented and complexity metrics.

If we examine Table 6.2, it seems clear that Law I (continuous change) and VI (continuing growth) have been validated by all the studies. Law II (increasing complexity), IV (conservation of organizational stability) and V (conservation of familiarity) have been invalidated for most of the cases. These three laws are related to the growth pattern of the project. Software projects whose growth accelerates over time do not fulfill these three laws.

All the mentioned studies are based either in qualitative considerations, or in empirical evaluation of size and complexity over time. However, Fernandez-Ramil *et al.* think that the laws should be validated through simulation as well as through empirical studies [Fernández-Ramil et al. 2008]. The laws are coupled, so the effects of one law are not independent of the effects of the rest of laws, but empirical studies cannot

isolate each law to verify it. Simulation can take into account this coupling between the laws. Simulation was already attempted in the nineties [Chatters et al. 2000; Wernick and Lehman 1999] in the FEAST project. More recently Smith *et al.* [2005] have simulated the evolution of libre software. According to them, the laws of software evolution are a natural language expression of empirical data, and can be mapped to many different lower level empirical scenarios. The diversity of scenarios is the root of the discrepancy between different empirical studies.

6.3. The debate on metrics in confirming studies

The study by Israeli and Feitelson is the first one providing an extensive quantification of the laws of software evolution. The initial studies in the decade of the 2000s [Godfrey and Tu 2000; 2001; Robles et al. 2005] measured only size in lines of code. Israeli and Feitelson studied the size of Linux using lines of code, number of system calls, number of configuration options and number of functions. They also measured some other complexity metrics, and the maintainability index for estimating the quality of the system [Coleman et al. 1994].

The formulation of the laws does not include any mention of how to quantify them, even though Lehman proposed the laws based on empirical results that used a particular set of metrics, and therefore they had an implicit quantification. This lack of an explicit quantification of the laws is the source of the *anomaly* incident described at the beginning of this section. Lehman questioned the studies on Linux because they were using different metrics than the original software evolution studies. However, neither the laws themselves nor any publication on Lehman's theory of software evolution specify metrics, measurements or processes to evaluate or verify the laws.

In the early studies on software evolution, Lehman, Belady and others did not have direct access to the source code and the change records. They could gain access only to some datasets provided by companies, owners of the systems, with a small set of metrics.

Having only a single data point per release, Lehman decided to use releases as a pseudo-unit of time, formalized as *Release Sequence Number* (RSN). He decided to use the number of modules as the base unit for size as well.

These units have persisted in later works by Lehman and others [Lehman 1996b; Lehman et al. 1997; Ramil and Lehman 2000; Lehman et al. 2001], although some of the simulations in the FEAST project used calendar time rather than RSN [Chatters et al. 2000; Wernick and Lehman 1999]. Other studies have used Source Lines of Code (SLOC) as size metric [Godfrey and Tu 2001; Robles et al. 2005]. Therefore, the metrics used in some of the non-confirming studies were different to those of the original studies.

These differences were the base of Lehman's argument [2001], when he labeled the case of Linux [Godfrey and Tu 2000] as an anomaly, raising concerns about the comparability of studies, because they were using different metrics. In a previous work, Lehman *et al.* [1998a] had referred to the suitability of lines of code as a size metric for evolution studies:

> A lower level metric, lines of code (locs, or equivalently klocs) much beloved by industry, does not have semantic integrity in the context of system functionality. Moreover, when programming style directives are laid down in an organisation, the number of locs produced to implement a given function are very much a matter of individual programmer taste and habit.

Some studies indicate that the conflicting studies questioning the validity of the laws of software evolution are methodologically comparable to the original studies by Lehman, regardless of the metrics chosen. Nonetheless, both size metrics have been

shown to be comparable in in several studies, including some cases with very large datasets, containing thousands of software systems [Herraiz et al. 2006; Herraiz et al. 2007; Herraiz 2009].

The topic of time units in software evolution studies has attracted less research than the case of size metrics. To the extent of our knowledge, only two studies recommend to use calendar time rather than RSN when calendar time data are available [Barry et al. 2007; Vasa 2010].

7. WHAT HAS THE RESEARCH COMMUNITY FOUND ABOUT THE LAWS OF SOFTWARE EVOLUTION?

7.1. Research Question 1: Why did the laws of software evolution undergo so many modifications during their early life, but have not changed in the last 15 years?

The last modification of the laws of software evolution dates from 1996 [Lehman 1996b], while the main works about the validity of the laws were published later (see Section 6.2). There were some early empirical studies about the validity, though [Lawrence 1982; Pirzada 1988].

However Lehman did not change the laws because of the early empirical findings. He adapted the laws to the new development practices that evolved during the eighties and the nineties. He also took into account new concepts, such as feedback, to adapt the formulation of the laws [Lehman and Ramil 2003].

To address the problem of different software development, maintenance and releasing practices, Lehman coined the SPE scheme, modifying the laws, and making it explicit that they are only valid for a particular type of software, i.e., *E*-type.

Perry proposed the notion of domains in software evolution [Perry 1994]. Evolutionary behavior could differ for software in different domains. The need for domains is very similar to the reasons that led Lehman to propose the SPE classification scheme: not all software evolves in the same way. As a matter of fact, the SPE scheme was recently updated [Cook et al. 2006b] to clarify the differences between software evolution categories (*domains*).

The case of libre software is a paradigmatic example of a new domain that has been ignored in the definition of the laws and the SPE scheme. Prior to the appearance of the Internet, no software was developed by teams distributed across the globe, without regularly meeting in person, and communicating just through electronic channels. This environment is very different from the software engineering practices common during the times of the first versions of the laws.

However, although Lehman updated the SPE classification scheme [Cook et al. 2006b], he did not incorporate the notion of domain into the laws, or tried to adapt them to different software engineering practices. The laws kept exclusively referring to *E*-type software. Thus, as a summary, we can conclude that:

— During the first decades, the laws did not evolve as an answer to new empirical findings questioning their validity.
— The main modifications in the laws were due to changes in software development and maintenance practices and standards. The only source of modifications are publications from Lehman and collaborators. Proposals of modifications by other authors (such as [Pirzada 1988]) were explicitly rejected [Lehman and Ramil 2003].
— In modern publications by Lehman and collaborators, they have not taken into account recent studies about the validity for libre software. The most recent publication about the laws [Lehman and Fernández-Ramil 2006] still contains the same formulation that was published 10 years before[Lehman 1996b].

7.2. Research Question 2: Which laws have been confirmed in the research literature? Which laws have been invalidated?

In the beginnings of the software evolution research field, the laws could only be verified with very few case studies. Lehman extracted his conclusions mainly from the IBM OS/360 operating system project [Lehman 1980]. In addition to Lehman's empirical findings, the laws were also validated using mathematical models [Woodside 1980].

When the laws started to confront new empirical studies, the conclusions were that not all the laws were universally valid. The first to raise the issue of validity was Lawrence [1982]. For the laws, as formulated at the time, (see Tables 3 and 4.2), his data could only validate Law I (continuing change) and II (increasing complexity). Law III (statistically smooth evolution), IV (invariant work rate) and V (conservation of familiarity) could not be validated.

Similar results were obtained by Pirzada [1988]. He validated all the laws, but only for software that underwent a commercialization process. This kind of software evolves as predicted by the laws after it has been marketed. However, software developed in research and academic communities was evolving in a different way, and the laws could not explain its evolution.

During the nineties the results were promising. After the last version of the laws was published (see Table 5.1), Lehman *et al.* [1997] found empirical validation of all of them, except V (conservation of familiarity) and VII (declining quality). These two laws could not be validated because of the lack of empirical data. However, this does not mean that those laws were invalidated either. Other works could validate all the laws using simulation techniques [Chatters et al. 2000; Wernick and Lehman 1999]. Also, Turski's model [Turski 1996] was validated using industrial software as case study [Lehman et al. 1998b].

The case of libre software, already discussed in detail in Section 6, supports the validity of Law I (continuing change) and VI (continuous growth). It also shows that the second law (increasing complexity) is not valid. For other laws, the results are not consistent. A recent meta-study by Fernández-Ramil *et al.* [2008] agrees with with these results. Laws I and VI are valid, while Laws II to V could not be validated using empirical studies.

In summary, Lehman observed in the sixties and seventies that software needed to be changed and updated (continuous growth). These observations have remained valid over the years. Therefore Law I and VI seem to remain valid. The rest of the laws cannot be verified, although it is not clear that they are invalid either. In any case, it is interesting to highlight the case of the Law II (increasing complexity), which has been invalidated in many libre software empirical studies. According to Lehman, software changes degrade the structure of the system. This degradation increases the complexity of further changes. This is similar to Parnas' view of code decay and software aging [Parnas 1994]. However, empirical data shows that further changes are not more difficult, and thus, the complexity of software does not seem to increase with continuous changes.

7.3. Research Question 3: Which strategies, data and techniques can be used to validate the laws of software evolution?

The laws of software evolution have been studied at different levels of granularity (micro, macro), and using two main techniques: simulation and empirical studies.

The original works by Lehman were mainly empirical studies at a macro level [Lehman and Belady 1985]. Several works claiming the invalidity of the laws were also empirical studies at a macro level [Pirzada 1988; Godfrey and Tu 2000].

These invalidating studies measured size over time, and found that the curve of size over time was growing with superlinear profiles [Godfrey and Tu 2000; Robles et al. 2005]. That is, growth was accelerating. As it was said, this behavior is incompatible with the laws of software evolution. The most notable case of this kind of growth was the Linux kernel. However, when this very same case was analyzed in detail at a micro level, the superlinear growth curve was found to be a set of linear segments [Israeli and Feitelson 2010]. The aggregation of these segments produced a superlinear growth curve. However, individual modules were not growing with acceleration. The results of this micro level study about Linux [Israeli and Feitelson 2010] were in conflict with the previous macro studies.

This difference among different levels of granularity was also observed by Gall *et al*. [1997], who were the first ones to suggest the study of software evolution at different levels of granularity. These differences in the level of the study are known to have caused similar discrepancies in other research areas, and in some works in empirical software engineering [Posnett et al. 2011].

Another dimension of the validation of the laws is the approach followed for the validation process. We have presented two main classes of studies: simulation and empirical studies.

The first simulation work can be dated back to 1980, when Woodside published the first model based on the laws of software evolution [Woodside 1980] (later republished as [Woodside 1985]). More recently, the model by Turski [Turski 1996] was used to validate the laws using simulation [Lehman et al. 1998b]. In general, the FEAST project was the main source of simulation studies [Chatters et al. 2000; Lehman and Wernick 1998; Wernick and Lehman 1999]. Most of these simulation studies have been done at a macro level, considering overall properties of the software projects over time. However, some models are also of a micro nature, studying the software development and maintenance processes [Smith et al. 2005].

The laws of software evolution are tightly coupled. It is difficult to isolate the effects of one law while discarding the rest. This makes it more difficult to validate the laws using empirical studies. In fact, all empirical studies reviewed here tried to validate (or invalidate) the laws one by one, without considering any interaction between them. However, with simulation studies, it is possible to study how the laws change all at the same time, and to take into account the influence of one law over the others.

On the other hand, the advantage of empirical studies is the scale of the study. It is possible to study large amounts of software projects using an empirical approach (e.g. Koch studied more than 8500 software projects [Koch 2007]). To our knowledge, there are no large scale simulation studies.

Summarizing, what we have found is that:

— The validation process is done through simulation and empirical studies.
— Empirical studies yield different (and conflicting) results if they analyze evolution at different levels of granularity (using the same case studies).
— Simulation studies are usually of small scale, that is, they are empirically validated with only a few software projects. There is a lack of large scale simulation studies.
— Empirical studies can be of large scale, but at a macro level. There are no microlevel large scale empirical studies.

7.4. Research Question 4: Which kind of software projects, and under what conditions, fulfill the laws of software evolution?

Initially, Lehman conceived the laws of software evolution as applicable to large software systems [Lehman 1974]. Those systems are developed by teams organized in sev-

eral managerial levels, and they count with a large user base which provides feedback about the quality of the system, and demands new features.

Lehman later improved the definition of "large software system", proposing the SPE scheme [Lehman 1980]. This scheme classifies software in three categories. Only one of those categories, *E*-type, is described by the laws.

Other empirical studies found that software evolution is different for different kinds of software projects [Pirzada 1988]. In particular, only software that undergoes a commercialization process was found to follow the laws.

Koch found differences in the evolution of libre software projects of different sizes [Koch 2007]. Small libre software projects that are developed by a single person or a small group of people fulfill the laws. However, large software projects do not follow them. These projects have a large number of participants and a high asymmetry in the distribution of work among participants.

Other studies have analyzed the quality of the evolution of different kinds of software projects [Mockus et al. 2002], but not explicitly addressed the applicability of the laws. The main conclusion of [Mockus et al. 2002] is that there are significant differences in the quality and evolution of libre and non-libre software.

To summarize, we can only conclude that software projects of different kinds evolve differently. Despite the intense research, especially in the case of libre software in recent years, we have not found any other concluding remark.

8. WHAT'S NEXT? CHALLENGES AND DIRECTIONS FOR SOFTWARE EVOLUTION RESEARCH

In his 1974 lecture, Lehman suggested the study of software evolution with an empirical approach based on statistical methodologies [Lehman 1974]. At that time, the approaches suggested by Lehman were very difficult to implement in practice. However, the current research environment makes that much easier. Based on the findings of the previous sections, and on over 40 years of software evolution research literature, we suggest some challenges and directions for the next years of software evolution, which would help to achieve Lehman's original goal.

8.1. The importance of replication for empirical studies of software evolution

Public availability of the data used for empirical studies is crucial. A theory of software evolution must be based on empirical results, verifiable and repeatable, and made on a large scale, so that conclusions with statistical significance can be achieved [Sjøberg et al. 2008]. If software evolution is analyzed with data that is not available to third parties, it cannot be verified, repeated and replicated. It is dangerous to build a theory on top of empirical studies that do not fulfill those requirements.

Empirical studies of software evolution should conform to the guidelines suggested for empirical software engineering [Kitchenham et al. 2002; Kitchenham et al. 2008]. The entry barrier for repeatable empirical studies can be lowered using reusable datasets such as FLOSSMetrics [Herraiz et al. 2009], FLOSSMole [Howison et al. 2006], the Qualitas Corpus [Tempero et al. 2010] or the UCI Sourcerer Dataset [Lopes et al. 2010]. These datasets aim to achieve replicability of empirical studies of software development and evolution. Replicability is a concern recently raised in the empirical software engineering research community [González-Barahona and Robles 2012], with many authors highlighting its potential benefits [Robles and German 2010; Shull et al. 2008; Brooks et al. 2008; Barr et al. 2010].

However, Kitchenham warns about some of the problems that may arise with replication and sharing [Kitchenham 2008],in particular when reusing the so-called *laboratory packages*. If these packages are only gathered once and reused many times, that would amplify the probable errors that occur during the gathering phase. The avail-

ability of datasets does not remove the need for new datasets, that can be used to test the correctness of previous results. In any case, this does not invalidate the previous argument: software evolution studies must be replicable, either by reusing third party datasets and tools, or by making their data and/or tools publicly available.

8.2. Statistical methodologies for software evolution

In the seventies, the methodology proposed by Lehman to study software evolution was primarily based on a statistical study of different product and process metrics. He explicitly suggested the use of regression techniques, autocorrelation plots and time series analysis for the study of software evolution, based on the idea of feedback driven evolution [Lehman 1974].

Although time series analysis has not been a popular technique in the research community, some initial results show that it is a promising technique that should be explored [Herraiz et al. 2007; Herraiz et al. 2008].

One possible cause that could explain the lack of research using this approach is that this kind of analysis requires historical information. If we focus in the case of libre software, it is relatively easy to retrieve the source code releases of a project, or even of a large sample of projects, thanks to the availability of corpus such as *Qualitas* [Tempero et al. 2010] and *Sourcerer* [Lopes et al. 2010].

However historical information other than source code is harder to obtain. For instance, version control systems are very heterogeneous, from a semantic point of view. It is difficult to compare historical information extracted from different kinds of version control repositories. For large samples of projects, the variability in the different repositories used by each project makes it difficult to compare and aggregate the information. These problems have been recently addressed by some research projects such as SQO-OSS [Gousios and Spinellis 2009], FLOSSMetrics [Herraiz et al. 2009] and FLOSSMole [Howison et al. 2006]. The tools and datasets published by these projects should lower the entry barrier to apply statistical techniques using richer and more complex historical information. A recent survey [Kagdi et al. 2007] reviewed the works that study software evolution from a mining software repositories perspective.

8.3. Quantification and formalization of the laws

The recent research activity around the validity of the laws for libre software is an example of the ambiguity of applying and measuring the laws of software evolution. According to Fernández-Ramil *et al.* [2008], the laws cannot be quantified in a unique manner. Each law accepts different mappings to lower level metrics. In the opinion of Israeli and Feitelson [2010], this lack of formalization of the laws is the cause of discrepancies in the validation studies using Linux as a case study [Godfrey and Tu 2000; Robles et al. 2005].

The SPE scheme is insufficient for the formalization of the laws. The next version of the laws of software evolution should incorporate Perry's dimensions, or some other classification scheme adapted to the richer current software environment. It is doubtful that the laws can be universal, except perhaps for Law I and VI. The rest of them should be specialized for different kinds of software, and different software development and maintenance practices and standards.

Mens *et al.* [2005] also recommend a formalization of the theory of software evolution. An area that is lacking more contact with software evolution is formal methods, where specifications are typically not allowed to change. Mens *et al.* defend an integration of formal methods and software evolution research. This would foster that software evolution and formal methods are better accepted by software developers, who deal on a daily basis with the software evolution phenomenon.

9. RECOMMENDED READINGS

Table 9 shows a classification of a reduced selection of papers on software evolution, since the early papers by Lehman to the most recent empirical studies leading to future research. The theory of software evolution was mainly addressed by Lehman and close collaborators up to the nineties. During the last years, several papers question the future of software evolution as a field, proposing challenges and directions for research [Mens et al. 2005; Godfrey and German 2008; Antoniol 2007].

The first paper questioning the validity of the laws for libre software [Godfrey and Tu 2000] has brought several other studies on the topic, with contradictory results [Robles et al. 2005; Koch 2007; Israeli and Feitelson 2010]. The lack of formalization of the laws and their ambiguity makes it harder to perform meta-analyses of these studies to extract common conclusions [Fernández-Ramil et al. 2008].

Simulation and modeling research of software evolution is another approach fostered by Lehman and collaborators. It was one of the main areas of the FEAST project in the nineties. Although there are several publications on simulating software evolution [Kahen et al. 2001; Wernick and Lehman 1999; Chatters et al. 2000], the number of studies addressing the simulation of the evolution of libre software is so far insufficient [Smith et al. 2005], and to the extent of our knowledge no study has tried to validate (or invalidate) the laws of software evolution for the case of libre software using simulation.

Finally, as stated previously, the software evolution literature is vast. Table 9 may help to focus on the main readings to those starting research on the field. These papers are important to understand the historical evolution of the laws, and are must-reads for any researcher new to the field. In addition to the works already mentioned in the above paragraphs, Table 9 also includes Lehman's seminal papers: the different works presenting the laws of software evolution [Lehman 1974; 1978; 1980; 1996b], the SPE classification scheme (and its modification [Lehman 1980; Cook et al. 2006b]), the principle of software uncertainty [Lehman 1989] and the notion of feedback driven software evolution [Lehman 1996a].

10. CONCLUSIONS

Software engineering still lacks a completely solid scientific basis. The laws of software evolution were one of the first approaches to define a theory of software systems that provides such a scientific basis. However, they have faced many controversial analysis since their inception, and now it seems clear that their validity, although proven in many cases, is not universal.

The laws were designed with *change* in mind. Software cannot be immutable (or at least, *E*-type software), and the laws have been shown to need changes themselves as time passed. Their formulation had to be adapted several times since they were first defined, to take into account new software development and maintenance practices and standards, so that they did not become obsolete. However, these changes to the laws are not enough. The overall software evolution theory must be adapted to face another fundamental principle: not only is software not immutable, but the way it is produced is not immutable either.

With respect to how the validation studies are conducted, the original statistical approach suggested by Lehman is now becoming possible because of the myriad of publicly available software repositories. Precisely because of this, the laws have to confront more studies, some of them questioning their validity.

However, there is one important aspect that current research is perhaps missing: a more general approach to invariants of software evolution. From this point of view, the important question is not whether the laws are universal, since their history has

Table VI. Classification of the main research literature on software evolution of the last decades

		<1980	1980-89	1990-99	>1999
Theory of software evolution	Lehman	[Belady and Lehman 1971] [Lehman 1974] [Lehman 1978]	[Lehman 1980] [Lehman 1989]	[Lehman 1996a] [Lehman 1996b] [Lehman and Ramil 1999]	[Cook et al. 2006b] [Lehman and Ramil 2001a]
	Others			[Perry 1994]	[Rajlich 2000] [Mens et al. 2005] [Antoniol 2007] [Godfrey and German 2008]
Validation or invalidation	Empirical	[Kitchenham 1982] [Lawrence 1982]	[Pirzada 1988]	[Gall et al. 1997] [Lehman et al. 1998b] [Bennett et al. 1999] [Kemerer and Slaughter 1999]	[Raml et al. 2000] [Fernández-Ramil et al. 2008] [Godfrey and Tu 2000] [Robles et al. 2005] [Koch 2007] [Israeli and Feitelson 2010]
	Simulation	[Belady and Lehman 1976]	[Woodside 1980]	[Wernick and Lehman 1999] [Turski 1996]	[Chatters et al. 2000] [Lehman et al. 2001] [Kahen et al. 2001] [Lehman et al. 2002] [Smith et al. 2005]

already shown how the laws themselves needed several changes. What is needed is a revamping of the laws, a quest for invariants in the evolution of software, a formal classification of invariants for the different kinds of software projects that can be currently found, and the expression of those invariants in precise ways that can be shown valid, or refuted, by empirical analysis.

Had Lehman started his research today, with the rich environment that we enjoy, he probably would have not tried to validate the old laws, but would have looked for invariant properties using new statistical and empirical approaches, classifying projects by applicability of the laws as he did with the SPE scheme, and obtaining models that would help in the development and maintenance of software. Sadly, Lehman is no longer among us to lead this research stream [Canfora et al. 2011]. The best tribute we can do to honor his work is to study modern programming processes as he did in the sixties and seventies to help developers, managers and users in order to produce better software.

REFERENCES

G. Antoniol. 2007. Requiem for software evolution research: a few steps toward the creative age. In *Proceedings of the International Workshop on Principles of Software Evolution*. ACM, 1–3.

William Aspray. 1993. Meir M. Lehman, Electrical Engineer, an oral history. IEEE History Center. (1993). Rutgers University, New Brunswick, NJ, USA.

F. T. Baker. 1972. Chief programmer team management of production programming. *IBM Systems Journal* 11, 1 (1972), 56 –73.

Earl Barr, Christian Bird, Eric Hyatt, Tim Menzies, and Gregorio Robles. 2010. On the shoulders of giants. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research (FoSER '10)*. ACM, New York, NY, USA, 23–28.

E.J. Barry, C.F. Kemerer, and S.A. Slaughter. 2007. How software process automation affects software evolution: a longitudinal empirical analysis. *Journal of Software Maintenance and Evolution: Research and Practice* 19, 1 (2007), 1–31.

Andreas Bauer and Markus Pizka. 2003. The Contribution of Free Software to Software Evolution. In *Proceedings of the International Workshop on Principles of Software Evolution*. IEEE Computer Society, Helsinki, Finland.

L. A. Belady. 1979. On software complexity. In *Proceedings of the Workshop on Quantitative Software Models for Reliability*. IEEE Computer Society, Kiamesha Lake, NY, USA, 90–94.

L. A. Belady. 1985. On software complexity. In *Program Evolution. Processes of Software Change*, M. M. Lehman and L. A. Belady (Eds.). Academic Press Professional, Inc., San Diego, CA, USA, 331–338.

L. A. Belady and M. M. Lehman. 1971. Programming System Dynamics or the Metadynamics of Systems in Maintenance and Growth. *Research Report RC3546, IBM* (1971).

L. A. Belady and M. M. Lehman. 1976. A Model of Large Program Development. *IBM Systems Journal* 15, 3 (1976), 225–252.

K. Bennett, E. Burd, C. Kemerer, M. M. Lehman, M. Lee, R. Madachy, C. Mair, D. Sjoberg, and S. Slaughter. 1999. Empirical Studies of Evolving Systems. *Empirical Software Engineering* 4, 4 (1999), 370–380.

G. Benyon-Tinker. 1979. Complexity measures in an evolving large system. In *Proceedings of the Workshop on Quantitative Software Models for Reliability*. IEEE Computer Society, Kiamesha Lake, NY, USA, 117–127.

A. Brooks, M. Roper, M. Wood, J. Daly, and J. Miller. 2008. Replication's Role in Software Engineering. In *Guide to Advanced Empirical Software Engineering*, Forrest Shull, Janice Singer, and Dag I. K. Sjberg (Eds.). Springer London, 365–379.

Frederick P. Brooks. 1978. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Gerardo Canfora, Darren Dalcher, David Raffo, Victor R. Basili, Juan Fernández-Ramil, Václav Rajlich, Keith Bennett, Liz Burd, Malcolm Munro, Sophia Drossopoulou, Barry Boehm, Susan Eisenbach, Greg Michaelson, Darren Dalcher, Peter Ross, Paul D. Wernick, and Dewayne E. Perry. 2011. In memory of Manny Lehman, "Father of Software Evolution". *Journal of Software Maintenance and Evolution: Research and Practice* 23, 3 (2011), 137–144.

B. W. Chatters, M. M. Lehman, J. F. Ramil, and P. Wernick. 2000. Modelling a software evolution process: a long-term case study. *Software Process: Improvement and Practice* 5, 2-3 (2000), 91–102.

C.K.S. Chong Hok Yuen. 1980. *A Phenomenology of Program Maintenance and Evolution*. Ph.D. Dissertation. Imperial College, London.

D. Coleman, D. Ash, B. Lowther, and P. Oman. 1994. Using metrics to evaluate software system maintainability. *IEEE Computer* 27, 8 (Aug. 1994), 44–49.

Stephen Cook, Rachel Harrison, Meir M. Lehman, and Paul Wernick. 2006a. Evolution in Software Systems: Foundations of the SPE Classification. In *Software Evolution and Feedback. Theory and Practice*, Nazim H. Madhavji, Juan Fernández-Ramil, and Dewayne E. Perry (Eds.). Wiley, 95–130.

Stephen Cook, Rachel Harrison, Meir M. Lehman, and Paul Wernick. 2006b. Evolution in Software Systems: Foundations of the SPE Classification Scheme. *Journal of Software Maintenance and Evolution: Research and Practice* 18, 1 (2006), 1–35.

EBSE. 2010. Template for a Systematic Literature Review Protocol. http://www.dur.ac.uk/ebse/resources/templates/SLRTemplate.pdf. (2010). http://www.dur.ac.uk/ebse/resources/templates/SLRTemplate.pdf

Juan Fernández-Ramil, Daniel Izquierdo-Cortazar, and Tom Mens. 2009. What Does It Take to Develop a Million Lines of Open Source Code? In *Open Source Ecosystems: Diverse Communities Interacting*, Cornelia Boldyreff, Kevin Crowston, Bjrn Lundell, and AnthonyI. Wasserman (Eds.). IFIP Advances in Information and Communication Technology, Vol. 299. Springer Berlin Heidelberg, 170–184.

Juan Fernández-Ramil, Angela Lozano, Michel Wermelinger, and Andrea Capiluppi. 2008. Empirical Studies of Open Source Evolution. In *Software Evolution*, Tom Mens and Serge Demeyer (Eds.). Springer, 263–288.

Harald Gall, Mehdi Jazayeri, René Klösch, and Georg Trausmuth. 1997. Software Evolution Observations Based on Product Release History. In *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society, 160–170.

Michael W. Godfrey and Daniel M. German. 2008. The Past, Present, and Future of Software Evolution. In *International Conference in Software Maintenance (ICSM) Frontiers of Software Maintenance*.

Michael W. Godfrey and Qiang Tu. 2000. Evolution in Open Source Software: A Case Study. In *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society, Washington, DC, USA, 131–142.

Michael W. Godfrey and Qiang Tu. 2001. Growth, Evolution, and Structural Change in Open Source Software. In *Proceedings of the International Workshop on Principles of Software Evolution*. Vienna, Austria, 103–106.

Jesús González-Barahona and Gregorio Robles. 2012. On the reproducibility of empirical software engineering studies based on data retrieved from development repositories. *Empirical Software Engineering* 17, 1 (2012), 75–89.

G. Gousios and D. Spinellis. 2009. A platform for software engineering research. In *Mining Software Repositories, 2009. MSR '09. 6th IEEE International Working Conference on*. 31–40. DOI:http://dx.doi.org/10.1109/MSR.2009.5069478

Israel Herraiz. 2009. A statistical examination of the evolution and properties of libre software. In *Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM)*. IEEE Computer Society, 439–442.

Israel Herraiz, Jesus M. Gonzalez-Barahona, and Gregorio Robles. 2007. Towards a theoretical model for software growth. In *International Workshop on Mining Software Repositories*. IEEE Computer Society, Minneapolis, MN, USA, 21–30.

Israel Herraiz, Jesus M. Gonzalez-Barahona, and Gregorio Robles. 2008. Determinism and Evolution. In *Proceedings of the International Working Conference on Mining Software Repositories*. ACM, Leipzig, Germany, 1–10.

Israel Herraiz, Jesus M. Gonzalez-Barahona, Gregorio Robles, and Daniel M. German. 2007. On the prediction of the evolution of libre software projects. In *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society, Paris, France, 405–414.

Israel Herraiz, Daniel Izquierdo-Cortazar, Francisco Rivas-Hernandez, Jesus M. Gonzalez-Barahona, Gregorio Robles, Santiago Dueñas-Dominguez, Carlos Garcia-Campos, Juan Francisco Gato, and Liliana Tovar. 2009. FLOSSMetrics: Free / Libre / Open Source Software Metrics. In *Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE Computer Society.

Israel Herraiz, Gregorio Robles, Jesus M. Gonzalez-Barahona, Andrea Capiluppi, and Juan F. Ramil. 2006. Comparison between SLOCs and number of files as size metrics for software evolution analysis. In *Proceedings of the European Conference on Software Maintenance and Reengineering*. IEEE Computer Society, Bari, Italy, 203–210.

James Howison, Megan Conklin, and Kevin Crowston. 2006. FLOSSMole: a collaborative repository for FLOSS research data and analyses. *International Journal of Information Technology and Web Engineering* 1, 3 (July-September 2006), 17–26.

Ayelet Israeli and Dror G. Feitelson. 2010. The Linux kernel as a case study in software evolution. *Journal of Systems and Software* 83, 3 (2010), 485 – 501.

H. Kagdi, M.L. Collard, and J.I. Maletic. 2007. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution: Research and Practice* 19, 2 (2007), 77–131.

G. Kahen, M. M. Lehman, J. F. Ramil, and P. Wernick. 2001. System dynamics modelling of software evolution processes for policy investigation: Approach and example. *Journal of Systems and Software* 59, 3 (2001), 271 – 281. DOI:http://dx.doi.org/10.1016/S0164-1212(01)00068-1

C. F. Kemerer and S. Slaughter. 1999. An empirical approach to studying software evolution. *IEEE Transactions on Software Engineering* 25, 4 (1999), 493–509.

Barbara A. Kitchenham. 1982. System evolution dynamics of VME/B. *ICL Technical Journal* 3 (1982), 43–57.

Barbara A. Kitchenham. 2008. The role of replications in empirical software engineering: a word of warning. *Empirical Software Engineering* 13, 2 (2008), 219–221.

Barbara A. Kitchenham, Hiyam Al-Khilidar, Muhammed Babar, Mike Berry, Karl Cox, Jacky Keung, Felicia Kurniawati, Mark Staples, He Zhang, and Liming Zhu. 2008. Evaluating guidelines for reporting empirical software engineering studies. *Empirical Software Engineering* 13, 1 (2008), 97–121.

Barbara A. Kitchenham and S. Charters. 2007. *Guidelines for performing Systematic Literature Reviews in Software Engineering*. Technical Report EBSE-2007-01. Keele University.

Barbara A. Kitchenham, Shari Lawrence Pfleeger, Lesley M. Pickard, Peter W. Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. 2002. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering* 28, 8 (August 2002), 721–734.

Stefan Koch. 2005. Evolution of Open Source Software Systems - A Large-Scale Investigation. In *Proceedings of the International Conference on Open Source Systems*. Genova, Italy.

Stefan Koch. 2007. Software evolution in open source projects—a large-scale investigation. *Journal of Software Maintenance and Evolution: Research and Practice* 19, 6 (2007), 361–382.

M. J. Lawrence. 1982. An examination of evolution dynamics. In *Proceedings of the International Conference on Software Engineering*. IEEE Computer Society Press, Tokyo, Japan, 188–196.

M. M. Lehman. 1974. Programs, Cities, Students: Limits to Growth? (1974). Inaugural lecture, Imperial College of Science and Technology, University of London.

M. M. Lehman. 1978. Laws of Program Evolution-Rules and Tools for Programming Management. In *Proceedings of Infotech State of the Art Conference, Why Software Projects Fail*.

M. M. Lehman. 1979. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software* 1 (1979), 213 – 221.

M. M. Lehman. 1980. Programs, life cycles, and laws of software evolution. *Proc. IEEE* 68, 9 (1980), 1060–1076.

M. M. Lehman. 1984. Program Evolution. *Information Processing and Management* 20, 1-2 (1984), 19–36.

M. M. Lehman. 1985a. Program Evolution. In *Program Evolution. Processes of Software Change*, M. M. Lehman and L. A. Belady (Eds.). Academic Press Professional, Inc., San Diego, CA, USA, 9–38.

M. M. Lehman. 1985b. The Programming Process. In *Program Evolution. Processes of Software Change*, M. M. Lehman and L. A. Belady (Eds.). Academic Press Professional, Inc., San Diego, CA, USA, 39–84.

M. M. Lehman. 1985c. Programs, Cities, Students: Limits to Growth? In *Program Evolution. Processes of Software Change*, M. M. Lehman and L. A. Belady (Eds.). Academic Press Professional, Inc., San Diego, CA, USA, 133–164.

M. M. Lehman. 1989. Uncertainty in computer application and its control through the engineering of software. *Journal of Software Maintenance: Research and Practice* 1, 1 (1989), 3–27.

M. M. Lehman. 1990. Uncertainty in Computer Application. *Communications of ACM* 33, 5 (1990), 584–586. Technical letter.

Meir. M. Lehman. 1991. Software engineering, the software process and their support. *The Software Engineering Journal* 6, 5 (September 1991), 243–258.

M. M. Lehman. 1996a. Feedback in the software evolution process. *Information and Software Technology* 38, 11 (1996), 681 – 686.

M. M. Lehman. 1996b. Laws of Software Evolution Revisited. In *Proceedings of the European Workshop on Software Process Technology*. Springer-Verlag, London, UK, 108–124.

M. M. Lehman. 1998. Software's future: Managing evolution. *IEEE Software* 15, 1 (1998), 40–44.

M. M. Lehman and L. A. Belady. 1985. *Program evolution. Processes of software change*. Academic Press Professional, Inc., San Diego, CA, USA.

Meir. M. Lehman and Juan Fernández-Ramil. 2006. Software Evolution. In *Software Evolution and Feedback. Theory and Practice*, Nazim H. Madhavji, Juan Fernández-Ramil, and Dewayne E. Perry (Eds.). Wiley, 7–40.

M. M. Lehman, G. Kahen, and J. F. Ramil. 2002. Behavioural modelling of long-lived evolution processes: some issues and an example. *Journal of Software Maintenance and Evolution: Research and Practice* 14, 5 (2002), 335–351.

M. M. Lehman and F. N. Parr. 1976. Program Evolution and its impact on Software Engineering. In *Proceedings of the International Conference on Software Engineering*. IEEE Computer Society Press, Los Alamitos, CA, USA, 350–357.

Manny M. Lehman, Dewayne E. Perry, and Juan F. Ramil. 1998a. Implications of evolution metrics on software maintenance. In *Proceedings of International Conference on Software Maintenance*. IEEE Computer Society, 208–217.

M. M. Lehman, D. E. Perry, and J. F. Ramil. 1998b. On evidence supporting the FEAST hypothesis and the laws of software evolution. In *Proceedings of the International Software Metrics Symposium*. 84 –88.

M. M. Lehman and J. F. Ramil. 1999. The impact of feedback in the global software process. *Journal of Systems and Software* 46, 2–3 (1999), 123–134.

M. M. Lehman and J. F. Ramil. 2001a. An approach to a theory of software evolution. In *IWPSE '01: Proceedings of the 4th International Workshop on Principles of Software Evolution*. ACM Press, New York, NY, USA, 70–74. DOI:http://dx.doi.org/10.1145/602461.602473

Manny M. Lehman and Juan F. Ramil. 2001b. Rules and Tools for Software Evolution Planning and Management. *Annals of Software Engineering* 11, 1 (2001), 15–44.

M. M. Lehman and J. F. Ramil. 2002a. An Overview of Some Lessons Learnt in FEAST. In *Proceedings of the Workshop on Empirical Studies of Software Maintenance*.

M. M. Lehman and J. F. Ramil. 2002b. Software Uncertainty. In *Proc Soft-ware 2002*. Vol. 2311. 174+.

Meir M. Lehman and Juan F. Ramil. 2003. Software evolution: Background, theory, practice. *Inform. Process. Lett.* 88, 12 (2003), 33 – 44.

Manny M. Lehman, Juan F. Ramil, and U. Sandler. 2001. An Approach to Modelling Long-Term Growth Trends in Software Systems. In *International Conference on Software Maintenance*. IEEE Computer Society, Florence, Italy, 219–228.

Manny M. Lehman, Juan F. Ramil, P D. Wernick, D E. Perry, and W M. Turski. 1997. Metrics and Laws of Software Evolution - The Nineties View. In *Proceedings of the International Symposium on Software Metrics*.

M. M. Lehman and P. Wernick. 1998. System dynamics models of software evolution processes. In *Proceedings of International Workshop on the Principles of Software Evolution, IWPSE*. 20–24.

C. Lopes, S. Bajracharya, J. Ossher, and P. Baldi. 2010. UCI Source Code Data Sets. (2010). http://www.ics.uci.edu/~lopes/datasets/

Nazim H. Madhavji, Juan Fernández-Ramil, and Dewayne E. Perry (Eds.). 2006. *Software Evolution and Feedback. Theory and Practice*. Wiley.

Tom Mens and Serge Demeyer. 2008. *Software Evolution*. Springer, Berlin.

T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri. 2005. Challenges in software evolution. In *Proceedings of the International Workshop on Principles of Software Evolution*. 13 – 22.

Audris Mockus, Roy T. Fielding, and James D. Herbsleb. 2002. Two case studies of Open Source Software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology* 11, 3 (2002), 309–346.

Iulian Neamtiu, Guowu Xie, and Jianbo Chen. 2013. Towards a better understanding of software evolution: an empirical study on open-source software. *Journal of Software: Evolution and Process* 25, 3 (2013), 193–218.

David Lorge Parnas. 1994. Software Aging. In *Proceedings of the International Conference on Software Engineering*. Sorrento, Italy, 279–287.

Dewayne E. Perry. 1994. Dimensions of Software Evolution. In *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society, Washington, DC, USA, 296–303.

Dewayne E. Perry. 2006. A nontraditional view of the dimensions of software evolution. In *Software Evolution and Feedback. Theory and Practice*, Nazim H. Madhavji, Juan Fernández-Ramil, and Dewayne E. Perry (Eds.). Wiley, 41–51.

Shamin S. Pirzada. 1988. *A statistical examination of the evolution of the UNIX system*. Ph.D. Dissertation. Imperial College. University of London.

D. Posnett, V. Filkov, and P. Devanbu. 2011. Ecological inference in empirical software engineering. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. 362 –371.

A. Rainer and S. Gale. 2005. Sampling open source projects from portals: some preliminary investigations. In *Software Metrics, 2005. 11th IEEE International Symposium*. 10 pp. –27. DOI:http://dx.doi.org/10.1109/METRICS.2005.41

Vclav Rajlich. 2000. Modeling software evolution by evolving interoperation graphs. *Annals of Software Engineering* 9, 1-2 (March 2000), 235–248. http://springerlink.com/app/home/contribution.asp?wasp=f0506d5dd3f24ad5ae55ee545e913b11\& referrer=parent\& backto=issue,11,16; journal,6,12; browsepublicationsresults,132,2448;

V.T. Rajlich and K.H. Bennett. Jul 2000. A staged model for the software life cycle. *IEEE Computer* 33, 7 (Jul 2000), 66–71.

Juan F. Ramil and Meir M. Lehman. 2000. Metrics of Software Evolution as Effort Predictors - A Case Study. In *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society, 163–172.

J. F. Ramil, M. M. Lehman, and Goel Kahen. 2000. The FEAST Approach to Quantitative Process Modelling of Software Evolution Processes. In *Product Focused Software Process Improvement*, Frank Bomarius and Markku Oivo (Eds.). Lecture Notes in Computer Science, Vol. 1840. Springer Berlin / Heidelberg, 149–186.

Gregorio Robles, Juan Jose Amor, Jesus M. Gonzalez-Barahona, and Israel Herraiz. 2005. Evolution and Growth in Large Libre Software Projects. In *Proceedings of the International Workshop on Principles in Software Evolution*. Lisbon, Portugal, 165–174.

Gregorio Robles and Daniel M. German. 2010. Beyond Replication: An example of the potential benefits of replicability in the Mining of Software Repositories Community. In *Proceedings of the 1st International Workshop on Replication in Empirical Software Engineering Research (RESER)*.

Forrest Shull, Jeffrey Carver, Sira Vegas, and Natalia Juristo. 2008. The role of replications in Empirical Software Engineering. *Empirical Software Engineering* 13, 2 (2008), 211–218.

Nils T Siebel, Stephen Cook, Manoranjan Satpathy, and Daniel Rodrguez. 2003. Latitudinal and longitudinal process diversity. *Journal of Software Maintenance and Evolution: Research and Practice* 15, 1 (2003), 9–25.

Dag I. K. Sjøberg, Tore Dybå, Bente C. D. Anda, and Jo E. Hannay. 2008. Building Theories in Software Engineering. In *Guide to Advanced Empirical Software Engineering*, Forrest Shull, Janice Singer, and Dag I. K. Sjberg (Eds.). Springer London, 312–336.

Neil Smith, Andrea Capiluppi, and Juan F. Ramil. 2005. A study of open source software evolution data using qualitative simulation. *Software Process: Improvement and Practice* 10, 3 (2005), 287–300.

Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. 2010. Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies. In *Proceedings of the Asia Pacific Software Engineering Conference*.

Wladyslaw M. Turski. 1996. Reference Model for Smooth Growth of Software Systems. *IEEE Transactions on Software Engineering* 22, 8 (1996), 599–600.

Wladyslaw M. Turski. 2002. The Reference Model for Smooth Growth of Software Systems Revisited. *IEEE Transactions on Software Engineering* 28, 8 (2002), 814–815.

Rajesh Vasa. 2010. *Growth and change dynamics in open source software systems*. Ph.D. Dissertation. Swinburne University of Technology, Melbourne, Australia.

P. Wernick and M. M. Lehman. 1999. Software process white box modelling for FEAST/1. *Journal of Systems and Software* 46, 2-3 (1999), 193 – 201.

C. Murray Woodside. 1980. A mathematical model for the evolution of software. *Journal of Systems and Software* 1, 4 (1980), 337–345.

C. Murray Woodside. 1985. A mathematical model for the evolution of software. In *Program Evolution. Processes of Software Change*, M. M. Lehman and L. A. Belady (Eds.). Academic Press Professional, Inc., San Diego, CA, USA, 339–354.

Online Appendix to: The evolution of the laws of software evolution. A discussion based on a systematic literature review.

ISRAEL HERRAIZ, Technical University of Madrid, Spain
DANIEL RODRIGUEZ, University of Alcala, Madrid, Spain
GREGORIO ROBLES and JESUS M. GONZALEZ-BARAHONA, GSyC/Libresoft, University Rey Juan Carlos, Madrid, Spain

A. METHODOLOGY OF THE SYSTEMATIC LITERATURE REVIEW

This section describes the methodology followed to select and organize the final list of publications considered in the rest of the paper. We have conducted our literature survey according to the Systematic Literature Review (SLR) protocol and guidelines suggested in [Kitchenham and Charters 2007] and the EBSE website[8] [EBSE 2010].

A.1. Background and research questions

The background, research aim and objectives of the survey were already presented in Section 1. The research questions (RQ1-RQ4) guided our search and selection process, which can be summarized as: to identify, classify and analyze papers in the scientific literature discussing the laws of software evolution in order to find invariants, validating (and invalidating) studies to provide food for thought in future research work, especially in the libre software development field.

A.2. Data sources

As data sources, we used two kinds of compilations: general compilations (including some digital libraries), and specific compilations (some workshops, conferences and journals relevant in the field of software evolution, and publications by Lehman).

The general compilations we used are:[9]

(1) ACM Digital Library (http://www.acm.org/)
(2) Elsevier Science Direct (http://www.sciencedirect.com/)
(3) IEEE Xplore (http://ieeexplore.ieee.org/)
(4) SpringerLink (http://www.springerlink.com/)
(5) Wiley Interscience (http://onlinelibrary.wiley.com/)
(6) ISI Web of Science (http://apps.webofknowledge.com/)
(7) Google Scholar (http://scholar.google.com/)
(8) Scopus (http://www.scopus.com/)

These generic sources, together, probably reference all relevant publications in the software engineering field. Some of them are digital libraries, but some others are *meta-repositories*, indexing papers from different and disperse sources which certainly include most of the relevant journals, conference and workshop proceedings about software engineering.

[8]Evidence-Based Software Engineering http://www.dur.ac.uk/ebse/

[9]We also considered *Microsoft Academic Search* and *Sciverse Hub* but these sources were later discarded due to problems when using negation. Also, in the case of *Sciverse*, the literature is also included in *ScienceDirect*. Both issues are later discussed in the paper.

DOI: http://dx.doi.org/10.1145/0000000.0000000

The specific compilations, which we used because of their relevance in the field of software evolution, are:

— Proceedings of the *International Workshop on Principles of Software Evolution* (IWPSE), and of the ERCIM *Workshop on Software Evolution* (EVOL), which are jointly organized since 2009. They were obtained from the ACM Digital Library, IEEE Xplore library and Google Scholar.
— Proceedings of the *International Conference on Software Maintenance* (ICSM), obtained from the IEEE Xplore library, which contains those proceedings since 1988.
— Archives of the *Journal of Software Evolution and Process* (JSEP) published by Wiley, obtained from the Wiley Interscience digital library.
— Lehman's works. All the papers written by Meir M. Lehman were also included. For this, we used the complete listing of publications maintained at his Middlesex University web page.[10]

Although publications referenced in these specific compilations (e.g. IEEExplore) were probably all present in the general compilations (e.g. Google Scholar), we decided to include them for two reasons: (i) to ensure that no relevant work was missed, and (ii) to have a small set of publications to test the inclusion and exclusion criteria defined in the next section. In fact, since these specific compilations were small enough for a complete manual inspection, we used them while performing the SLR as a kind of control set, checking that the search methodology was accurate and correct, in addition to the background knowledge from the authors as well as following the relevant references from the papers.

A.3. Selection criteria

The selection criteria for our review consists of inclusion and exclusion criteria for the selected papers. The inclusion criteria are:

— To consider papers presenting studies, reviews or criticisms of the laws of software evolution, including all the early papers on software evolution that appeared before the laws were first stated.
— To consider also papers not explicitly mentioning the laws, but only when they discuss issues very similar to those considered by the laws.
— To consider only papers written in English, and published up to and including 2011.

The exclusion criterion is not to consider publications studying aspects of software evolution different from the laws themselves, or not dealing directly with them. This excludes, for example, literature related to maintainability studies which do not deal directly with the study of the laws of software evolution.

These criteria were used in pilot study in the specific compilations aforementioned, to help refining the search strategy. They were also checked against the results of the search strategy before determining the final set of studies.

A.4. Search strategy

Based on the previous criteria, the following specific queries were designed to conduct the search:

(Q1) "software evolution"
(Q2) "program evolution" AND NOT genetic
(Q3) law(s) AND "software evolution"
(Q4) law(s) AND "program evolution" AND NOT genetic

[10] http://www.eis.mdx.ac.uk/staffpages/mml/listing.html

Table VII. Fields and domains used for the queries in the different digital libraries

Digital Library	Domain	Digital Library	Domain
ISI Web of Knowledge	Computer Science Engineering	Google Scholar	Computer Science Maths Engineering
ACM Digital Library	-		
Wiley Interscience	-	IEEExplore	-
Elsevier ScienceDirect	Computer Science Engineering	SpringerLink	-
Microsoft Academic Search	Computer Science Engineering	Scopus	Computer Science Engineering
Sciverse	Computer Science Engineering		

The queries on some of the sources had to be constrained by field or domain. As each search service has different denominations for each field, we summarize the fields and domains selected in each one in Table A.4.

Notice that we searched both for the terms *software evolution* and *program evolution*. In 1980, Lehman [1980] used the term *laws of software evolution* for the first time. However, as late as 1985, we can observe occurrences of the term *program evolution* to denote the same concept [Lehman and Belady 1985]. In order to consider this duality in the terms, we decided to search for both for all the decades, even though the use of *software evolution* before the 80s and of *program evolution* after the 90s is highly unlikely.

However, the term *program evolution* is problematic, as it often appears on papers about evolutionary computation and genetic programming, which is a completely different field. To remove false positives, we decided to include *NOT genetic* so all those papers that include the word *genetic* would be discarded. There is a negligible probability of discarding legitimate papers, that is, papers which are about the laws of program evolution and contain the word *genetic*. Also the use of plural (law vs. laws) is irrelevant in some search cases.

After the first attempts with the mentioned queries, we decided to discard two of the compilations: Microsoft Academic Search and Sciverse. In Microsoft Academic Search, it was not possible to perform the negation operation operation in the queries, which implied too many false positives for the query Q4. Similarly, in Sciverse, the NOT operator did not seem to work correctly at the time we carried out our SLR. We considered that the contents indexed by Microsoft Academic Search and Sciverse Hub were probably already indexed by the conjunction all the rest of digital libraries, so we think we did not discard any relevant reference in this survey with the exclusion of these two compilations.

A.5. Selection procedure

The strategy for literature search was followed by two of the authors, which will be named 'assessors' in the rest of this section, in an independent manner. There were no noticeable differences between the papers obtained by both of them.

Both assessors performed a manual inspection of the titles and abstracts of the specific compilations to identify the papers that met the selection criteria. When a title or abstract seemed relevant for the survey, they were further explored the paper, reading the whole paper to decide its inclusion in the final list.

The experience obtained by the performance of this selection was used to determine the aforementioned search strategy. This search strategy was later followed on the complete general compilations, to identify the papers likely to be of interest. Once this shortlist was obtained, the papers that were not in the specific compilations were again manually inspected to have a final decision about their inclusion in the final list.

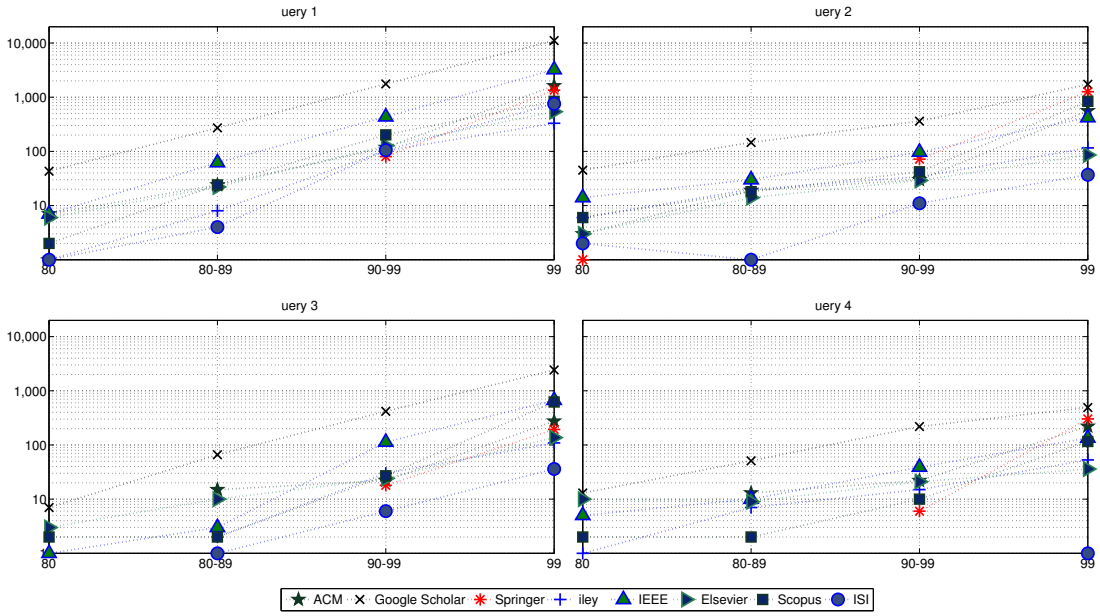

Fig. 2. Number of publications returned by the queries for each general compilation, by decade. Y axis (number of publications) is shown in logarithmic scale because of the wide differences between compilations and decades.

A small random subset of the papers obtained by using the search strategy, but not included in the final list, were examined in detail to ensure that the procedure was working, and that none deserved entering into the final list.

The final list of publications was organized into four periods: before 1980, 1980–1989, 1990–1990 and 2000–2011. In the following, we will refer to each one of these periods as decades, even though *before 1980* and *2010-2011* are not strictly decades.

A.6. Details of the selection procedure

Figure 2 shows the evolution of the number of publications obtained for each query for each of the general compilations. Two main observations can be made on the charts: (i) the growth of the number of publications has grown exponentially (as it can be observed from the straight line in the logarithmic plots); and (ii) the number of references to be considered in the field of software evolution is vast. This latter observation is also illustrated in Table A.6, which shows the maximum and minimum number of results for each query. For instance, query Q1 (*software evolution*) shows the total number of references found in Google Scholar was 13,175. If we bind the query to include an explicit mention of the term *law* or *laws* (query Q2, *software evolution* AND *law*), the number of references is reduced to 2,921, but still a large number to be considered. This evidences the need for a systematic survey of the field, which helps to organize, classify and extract conclusions from this large amount of literature.

However, many of these publications were not relevant for our survey. In order to discard non-relevant publications, we merged all references found, and sorted them by year. The assessors independently checked the lists for each year, deciding whether the paper was eligible for this survey. Eligibility decisions where primarily based on the title and abstract of the paper. If a paper had an apparently relevant title or abstract, its full content was also read, and the assessors decided whether the study was to be

Table VIII. Maximum and minimum number of results for each query. Name of the digital library in parentheses

Query	Maximum	Minimum
Q1	13,175 (Google Scholar)	442 (Wiley)
Q2	2,283 (Google Scholar)	51 (ISI)
Q3	2,921 (Google Scholar)	43 (ISI)
Q4	774 (Google Scholar)	1 (ISI)

Table IX. Papers written by Lehman according to the different compilations, and number of them including the terms *evolution* and *feedback* in their title or (in some compilations) abstract.

Digital Library	*Evolution*	*Feedback*	*Total*
ISI	28	15	34
Google Scholar	126	102	390
ACM	13	7	47
Wiley	5	4	5
IEEE Xplore	21	20	31
Elsevier	8	7	9
Springer	9	8	11
Scopus	36	22	44

included in the final list or not. The assessors used also their judgment and background knowledge about the field to select a reduced number of papers that were read before including or discarding them, even if the title did not seem to be relevant.

The compilation of Lehman's works deserves some specific comments. The compilation used lists 732 references, but we could only find 390 of them using Google Scholar, with the other general compilations returning even a lower number of results. To ensure that we missed no relevant paper written by Lehman, we combined the results of searching the compilations with the self-references published in his 1985's book [Lehman and Belady 1985], and all the self-references cited in those papers. Moreover, we bound the queries to include the terms *evolution* or *feedback*, because these are two main terms in the Lehman's literature on software evolution. This greatly reduced the number of papers to be considered. Table A.6 contains the number of papers found in each compilation for the different terms searched. In the case of his web page at Middlesex University, from the list of 732 papers, only 181 contain the word *evolution* on their title, and only 36 the word *feedback*.