Lehman's laws and the productivity of increments

Implications for productivity

Ramin Moazeni
Computer Science Department
University of Southern California
Los Angeles, U.S.A.
moazeni@usc.edu

Daniel Link
Computer Science Department
University of Southern California
Los Angeles, U.S.A.
dlink@usc.edu

Barry Boehm
Computer Science Department
University of Southern California
Los Angeles, U.S.A.
boehm@usc.edu

***Abstract*—Which are the consequences of Lehman's Laws of Software Evolution for the productivity of incrementally developed projects?**

The concept of Incremental Development Productivity Decline (IDPD), which deals with how the productivity of incrementally developed software develops over its increments, is introduced. It is explained how Lehman's Laws of Software Evolution apply to it and how maintenance and reuse are relevant to both.

Every Law of Software Evolution is discussed individually from a qualitative standpoint with regard to whether it could be a cause of IDPD. After that discussion, the overall situation is examined in light of how different courses of action cause which laws to apply different degrees of effects.

Keywords-software evolution, development productivity, incremental development.

I. INTRODUCTION

The increasing pace of change in software-intensive systems has increasingly shifted software engineering processes from single-pass, fixed-target development to a multiple-pass, evolutionary development process. Among other changes, this has changed the scope of software evolution to include not only relatively small maintenance changes to large operational systems, but also evolutionary development of similar-sized increments. This raises questions about the applicability of traditional principles or laws of software evolution to current and emerging forms of evolutionary development.

Understanding software evolution improves the ability to create processes for effective and consistent software development. Software evolution is defined as the process of going through the development, deployment and maintenance of software systems. Incremental Development Productivity Decline (IDPD) is a phenomenon by which, over the course of an incrementally developed project, the productivity of subsequent increments declines. The decline is due to maintenance and reuse factors such as previous-increment breakage and usage feedback, increased integration and testing effort.

The conventional productivity equation is Output per Input. While this is simple to understand, different metrics have been used to measure this in terms for software development. There is a host of different output metrics (SLOC, ESLOC and function points to name a few), but the available data from the observations limits their use. Typically, in software engineering, productivity is measured in SLOC per person-hour.

A. Incremental Development Productivity Decline

Incremental Development Productivity Decline (IDPD) is a phenomenon in which there is an overall decline in productivity of the increments. It should be clarified that for there to be IDPD, an overall decline includes cases where the decline between increments is not constant and where there can even be a rise in productivity between two given increments.

The IDPD factor is the percentage by which productivity decreases between two given increments. In order to be considered an increment for IDPD, an increment will have to contribute a significant amount of new functionality and add a significant amount of size over the previous one, i.e. it must not be less than a tenth of the size of the previous one and not be just a bug fix of the previous one. Otherwise it is considered part of the previous increment. It is important to understand the IDPD factor and its influence when doing incremental or evolutionary development. Ongoing research indicates that the magnitude of the IDPD factor may vary by type of

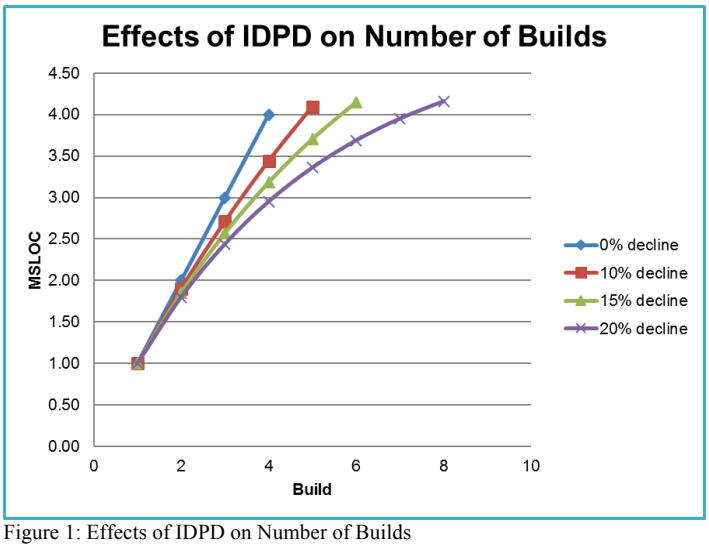

Figure 1: Effects of IDPD on Number of Builds

application (infrastructure software, such as operating systems and their extensions and virtual machines, having higher IDPDs since it tends to be tightly coupled and touches everything; applications software having lower IDPDs if it is architected to be loosely coupled), or by recency of the build (older builds may be more stable). The difference between a 10% IDPD and a 20% IDPD may appear small, but in practice it would a make a difference between completing in 5 builds vs. 8 builds. Fig. 1 shows the effects of IDPD on number of builds. It depicts a model relating productivity decline to number of builds needed to reach 8M SLOC Full Operational Capability.

B. Lehman's System Types

Lehman categorizes all software systems to fit one of three types: S, P or E. This classification was derived from a series of empirical observations Lehman made regarding large system development.

1) S-Type Systems

S-type or *static-type* systems are based on static, formal and verifiable sets of specifications with easy to understand solutions. These simple systems are generally not subject to evolution and therefore Lehman's laws don't apply.

2) P-Type Systems

P-type systems, or *practical-type* systems, can have their requirements defined precisely and formally, but the solution is not immediately apparent and its acceptability is dependent on the environment in which the program is being used. The key difference to S-type is that those are written to adhere to a static specification, but P-type programs are iteratively developed in order to improve their effectiveness.

3) E-Type Systems

The final type of system proposed by Lehman is the E-type, or *embedded-type*. E-type programs are defined as all programs that 'operate in or address a problem or activity of the real world'. Their evolution is a direct consequence and reflection of ongoing changes in a dynamic real world. Lehman's laws only describe the behavior of these projects, which mechanize human or societal activities and become part of the world they model [1],[2].

II. Lehman's Laws of Software Evolution

Lehman's laws describe how E-type software projects evolve over their releases. The original study Lehman and Belady conducted studied IBM software development of OS/360 in the 1970s. After this study, additional papers were written which expanded and amended the set of laws.

For non-trivial incremental development, the earlier increments will exist for some time before the project is finished (if there even is an end planned – some projects have continued to evolve). They will therefore either evolve or at least have a considerable age when the later increments are built. This applies to some, but not all programs observed here. These laws have been reviewed in the context of several major open source software projects, with some found to hold and others not [3].

A. General considerations

1) Parameters of the discussion

The following is a discussion about the individual Laws of Software Evolution as stated by Lehman in [1] and [2].

Some notes and caveats for the discussion are:

- The effort attributed to an increment also includes effort spent on previous increments during the time the new increment is developed.
- Usefulness is regarded as an indispensable part of what makes the quality of a piece of software sufficient. The authors posit that while other attributes than usefulness are part of the quality of software, developing a useless piece of software is pointless.
- There is no distinction made or intended between the terms "program" and "system". Reflecting the evolution of Software Engineering toward Systems and Software Engineering, Lehman's focus has shifted from software to systems over the 17 years that passed between the publications of [1] and [2], but the laws are intended for both, although the data support is based on software.

2) Maintenance and reuse

Maintenance can refer to upkeep effort that has to be expended on the codebase, its documentation, and the training of personnel and users.

Work on the codebase can consist of adding, modifying and deleting code.

There are different models for counting the code that is being added, modified and deleted. One option is of course to fully count the SLOC.

For cases of reuse (black box) and adaptation of code (white box), in some common models (such as COCOMO II, [6]) the effort is calculated by converting the code size to equivalent SLOC (ESLOC) based on how much effort will have to go into the integration of that code.

The COCOMO II reuse model needs to be laid out here because it is referenced by its maintenance model. It is a nonlinear reuse model that takes into account the understanding of the software (SU), the degree of assessment and assimilation necessary to integrate fully reused software modules and the unfamiliarity of the programmers with the software (UNFM). Additionally, the percentages of the existing software's design, code and previous integration effort that would be modi-

fied are taken into account as DM, CM and IM parameters. (The relatively complex formula is not reproduced here.)

For maintenance, COCOMO II uses the same model if the amount of the code base changed is less than or equal to 20%. Above that, a maintenance model is used that takes into account code size, SU and UNFM.

Documentation will have to be updated as needed. Training will have to be done to the degree needed in order to keep up productivity.

A "Reasoning" section is provided when a law's statement is in need of explanation.

B. Discussion of the individual Laws of Software Evolution

1) First law: Continuing Change

a) Statement

A program that is used and that as an implementation of its specification reflects some other reality, undergoes continual change or becomes progressively less useful. The change or decay process continues until it is judged more cost effective to replace the system with a recreated version [1]. Note that this "law" does not apply to many embedded software programs, which usefully control thermostats, carburetors, or most elevators because those programs tend to be the results of one-shot development efforts.

b) Application to IDPD

Incremental development means that increments build on each other. After each increment, there is a fully functional and tested program (or system – Lehman talks of "programs" in [1] and "systems" in [2], reflecting the widening scope of software engineering, the terms will therefore be used interchangeably for the duration of this discussion).

(That the program is used and reflects reality just means that it is an E-type system.)

The consequence of this law for the parts of the evolving systems to which it applies is that from the point in time that any increment gets released, the quality of the code base will decay unless effort goes into changing or replacing it. The change mentioned in the law can be any type of maintenance or even design change. It will result in the addition, deletion or modification of code. It is possible not to put any effort into this part of the existing codebase, but this will result in a reduction of quality and is therefore no a reasonable course of action for a rational actor. While no effort would be incurred in this case, money would be effectively spent due to the reduction of the quality of the overall system. Over time, the utility of the system would settle at a low point or degrade all the way to zero.

Assuming that the effort that can go toward a given increment is largely invariant (as stated by the Fourth Law further below), this means that a part of the effort for that increment will have to be directed toward keeping the existing code base useful. Together with the fact that the code base is growing over time as increments are added over time, this means that the part of the effort devoted to maintaining the quality of the code base will steadily increase and that the part that can be used to develop new code will steadily decrease. Even assuming that all other parameters stay the same (i.e. the productivity as expressed in SLOC per person-hour is the same for later increments), this will result in a lower new code output per time interval, therefore reducing the productivity in terms of new SLOC per person-hour.

The only situation in which this effort could be negligible is if the increments are developed in rapid succession or in cases where the reduction in quality is minimal.

c) Summary

The productivity in later increments will decline due to the rework and maintenance that has to be done on the earlier ones.

2) Second Law: Increasing Complexity

a) Statement

As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless effort is applied to maintain or reduce it [1].

b) Application to IDPD

As has been stated in the discussion of the First Law, if usefulness is to be retained, the existing code base will have to be constantly changed (i.e. maintained) as the system evolves.

The Second Law reflects the fact that with all maintenance, there are two layers of effort that need to be addressed if the quality of the whole system is to be kept equal:

1. The coding work itself
2. Integration of the coding work done into the system in terms of code integration, documentation, adaptation of the design and rework of its other sections.

The second kind of effort is addressed in this, the Second Law. As with the First Law, there is the option of not spending the effort, but the outcome will – again – be a loss of quality because the individual parts of the system will become less and less integrated, creating "rough patches" that are the results of adding code that is not well integrated, be it to fix deficiencies or to add functionality.

As in the case of the First Law, increments will require progressively more effort due to this, reducing the productivity from increment to increment.

If no effort is being directed toward reducing the complexity, the complexity will still add to the effort of the later increments by way of making their development more complex in turn because the more complex a previous increment becomes, the more complex it will become for the later increments to integrate with the previous ones.

c) Summary

Changing previous increments adds complexity, which in turn adds to the effort needed for the later increments.

3) Third Law: Fundamental Law of Program Evolution

a) Statement

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 invariances [1].

b) Reasoning

This law is a generalization of the fourth and fifth laws [1]. As with the other laws, laws 3-5 have large major exceptions, such as the effects of Y2K and Sarbanes-Oxley, and numerous individual exceptions, such as the effects of mergers and acquisitions on corporate infrastructure software.

c) Application to IDPD

This means that any change or variance in one system attribute will also be relevant for all others and that given enough information, the interdependencies of variables in a system should be predictable.

More specifically, the Fourth and Fifth Law state that organizations gravitate toward stability and that organizational dynamics and the need to maintain familiarity with the system impose upper limits on the output (that is, if that output is supposed to have any acceptable level of quality and the growth of the system is to be maintainable).

A fitting metaphor is therefore that of a cover that gets pulled in different directions.

For IDPD, this means that effort spent on one part of the system will directly influence how much effort can be spent on another. The same applies for different features of the system.

One possible example is that time spent on making the system more secure would take away from the time spent on improving the user interface. (This is because the law of Conservation of Organizational Stability says that the work rate per increment is invariant, see below.)

Another interpretation of the Third Law is that the evolution of a program is a predictable process where similar parameters should yield similar results. This means that the decline in productivity over the increments – IDPD – should also be predictable.

d) Summary

The amount of effort to be spent on any given increment is limited and there are interdependencies between effort parameters. Productivity should be predictable based on parameters.

4) Fourth Law: Conservation of Organizational Stability (Invariant Work Rate)

a) Statement

During the active life of a program the global activity rate in a programming project is statistically invariant [1].

b) Reasoning

Organizations are striving for stability and there is a point where "resource saturation" is reached and more resources will not make a difference [1].

c) Application to IDPD

General software engineering experience contradicts at least the literal meaning of this law: The activity rate can very well vary over the course of a project. This is evident because it is always possible to reduce the level of activity from the previous one (unless the initial one is already nil, but then there would be no actual programming project and the law would not apply). The law therefore needs to be interpreted.

A useful interpretation of this law is that there is an upper limit to the resources that can reasonably be committed to and the effort that can go into the development of a given increment. Beyond that limit, the effect of committing more resources or effort will either be little to no benefit or even detrimental. (This is in line with the well-known insight that "adding manpower to a late software project makes it later". [15])

Therefore, according to this law, there is no way for an organization to address the maintenance and integration effort that needs to go into existing increments by simply expending more resources in the way of hiring additional programmers. It is inevitable that this effort will take away from the activities that can be done on the later ones, reducing the productivity achievable for the new increment.

d) Summary

Beyond a certain upper limit, adding more resources or effort does not benefit the system in a meaningful way.

5) Fifth Law: Conservation of Familiarity (Perceived Complexity)

a) Statement

As an E-type system evolves all associated with it, developers, sales personnel, users, for example, must maintain mastery of its content and behavior to achieve satisfactory evolution. Ex-

cessive growth diminishes that mastery. Hence the average incremental growth remains invariant as the system evolves [2].

b) Application to IDPD

Everyone working on an incrementally developed project must maintain mastery of its content and behavior. The bigger the project gets, the more effort will have to be spent on this. Therefore, since the size of the project will grow over the course of its development, more and more effort will have to go into this and less will be able to go into new development.

If it is considered that the work rate is invariant, then there is an upper bound to the amount of effort that can go into any increment. Since the increment to be developed needs to make use of the existing increments, these will have to be maintained. The law of Conservation of Familiarity states that there is such a thing as "mental maintenance" which needs to be performed on the minds of the people working with the existing increments because otherwise, regardless of their quality, there is nobody who can integrate them.

c) Summary

Even if an increment may have solidified to the point that it needs little or no maintenance anymore, effort will have to go into the mental states of the people working with it. Excessive growth may be attempted for an increment, but the mastery of the system will have to keep up with the increments, so either the growth will not be able to take place or more training will have to be done after the increment, which reduces that increment's productivity and evens it out.

6) Sixth Law: Continuing Growth

a) Statement

The functional content of E-type systems must be continually increased to maintain user satisfaction over their lifetime [2].

b) Application to IDPD

All systems evaluated here are E-type systems. The systems are fully tested and functional after each increment. The lifetime of each increment runs until the overall system is decommissioned.

Therefore the functional content of every increment must be continually increased over the lifetime of the overall system if user satisfaction is to be achieved (which can reasonably be assumed to be the goal of the development of any system).

The effort on this will decrease the productivity of the current increment because it is done during the same time.

7) Seventh Law: Declining Quality

a) Statement

The quality of E-type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes [2].

b) Note

This law looks like a weaker phrasing of the one about Continuing Change that adds nothing of value: It's not relevant for software development and evolution whether the quality of an E-type system "appears" to be declining. The decisive question is whether it actually does. If it does not, then whether or not the systems are maintained and adapted is irrelevant. This law is therefore only of relevance if that section is changed to "will decline unless". If it is considered additionally that E-type systems respond to the environment by definition and that quality and being satisfactory are the same attributed, then this is the law about Continuing Change restated with different words.

c) The considerations about its application to IDPD are therefore the same as the ones regarding Continuing Change.

8) Eighth Law: Feedback System

a) Statement

E-type evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base [2].

b) Application to IDPD

A significant amount of parameters will have to be controlled in order to make incremental development a success. The productivity of a given increment and its quality is relevant to all following increments.

Lehman E-type systems don't exist in a vacuum. In order to stay relevant and useful to their environment, they have to react to the feedback from that environment, which they influence in turn. Reacting to the environment causes maintenance effort on the existing increments, which reduces the productivity of the later ones.

This further supports that effort going into the quality of earlier increments will improve the later increments but also take away productivity from them.

c) Summary

The parameters of all increments are relevant within the increments and to other increments.

III. FUTURE WORK

While this paper provides a background on IDPD and its application to Lehman's Laws of Software Evolution, the following areas need to be evaluated:

- Evaluate whether other sets of Software Engineering laws suggest that IDPD exists.

- Collect data in order to verify if the individual laws are supported in actual incremental projects.
- Map the extent of the relevance of individual laws to a given incrementally developed system to parametric cost drivers in order to predict the IDPD factor applying to that project.
- Research whether different types of incrementally developed systems, such as applications versus infrastructure software, have significantly different characteristics regarding how much the individual laws apply to them.

IV. CONCLUSION

Lehman's Laws indeed support IDPD.

When reviewing data collected from incrementally developed software and systems, a more detailed picture emerges:

- Laws 1, 2, and 6 are generally compatible with the overall trends in the IDPD data.
- Laws 3-5 imply that IDPD quantities are relatively constant from increment to increment. However, collected IDPD data rejects such a hypothesis.
- Laws 7-8 are external to the phenomenon involved in our IDPD data.

The overall insight gained from looking at all laws individual as well as in a group is that existing increments need to have maintenance performed on them, either directly by adding, modifying or deleting parts of their code, or by performing "mental maintenance" on the parties working with them. These efforts will then take away from the productivity that goes into new increments.

It is important to note that there are no reasonable ways to avoid the need for maintenance to occur. Any response to it will result in either more effort or a loss of quality for the system:

- Doing nothing and just focusing on the current increment to be developed will make the overall quality of the system degrade because the quality of the pre-existing increments will degrade (based on the application of the Law of Continuing Change to IDPD).
- Addressing the reduction in the quality of the existing increment by performing maintenance work on them will increase their complexity, which then in turn will have to be addressed (Law of Increasing Complexity).
- Any effort that is applied to one part of the system will take away from that which can be performed on the others (Fundamental Law of Program Evolution).
- The work rate cannot be changed at will, but only within limits (Conservation of Organizational Work Rate (Invariant Work Rate)).
- In order to keep existing increments useful, "mental maintenance" will have to be performed (Conservation of Familiarity (Perceived Complexity)).
- In order to keep existing increments useful, their functional content will have to be increased over their lifetime, which includes the development of later increments (Continuing Growth).

All this means that if a useful (i.e. qualitatively sufficient) system is to be built, the maintenance that will have to go into existing increments will take away productivity from the later ones. This is precisely what IDPD states. Lehman's Laws of Software Evolution therefore support the existence of IDPD individually and in combination.

REFERENCES

[1] M. M. Lehman, "Programs, Life Cycles, and Laws of Software Evolution", in Proc. of the IEEE, 1980, pp. 1060-1076. IEEE.

[2] M. M. Lehman et al., "Metrics and Laws of Software Evolution - The Nineties View", in Proceedings of the 4th International Symposium on Software Metrics, IEEE, 1997, pp. 20-32. IEEE.

[3] G. Xie, J. Chen, and I. Neamtiu, "Towards a Better Understanding of Software Evolution: An Empirical Study on Open Source Software," In *IEEE Conference on Software Maintenance (ICSM'09)*, September 2009.

[4] B. Boehm, et al., "Future Software Sizing Metrics and Estimation Challenges," 15th Annual Practical Systems and Software Measurement (PSM) Users' Group Conference, uly 2011

[5] T. Tan, Q. Li, B. Boehm, Y. Yang, M. He, R. Moazeni, "Productivity Trends in Incremental and Iterative Software Development," 3rd International Symposium on Empirical Software Engineering and Measurement, 2009

[6] B. Boehm, et al., "Software Cost Estimation with COCOMO II," Upper Saddle River: Prentice Hall, 2000

[7] Bennet, K., Rajlich, V., & Wilde, N. (2002). Software Evolution and the Staged Model of the Software Lifecycle. (M. Zelkowitz, Ed.) *Advances in Computers, 56*, pp. 1-54.

[8] IEEE 610.12. (1990). *Standard Glossary of Software Engineering Terminology.* New York: Internatonal Organization for Standardization and Institute of Electrical and Electronic Engineers. Retrieved August 1, 2010 from http://www.apl.jhu.edu/Notes/Hausler/web/glossary.html.

[9] ISO/IEC 12207, & IEEE 12207. (2008). *International Standard: Software Engineering - Software Life Cycle Processes.* New York: Internatlional Organization for Standardization and Institute of Electrical and Electronic Engineers.

[10] ISO/IEC 14764 , & IEEE 14764. (2006). *International Standard: Software Engineering -- Software Life Cycle Processes -- Software Maintenance.* New York: Internatonal Organization for Standardization and Institute of Electrical and Electronic Engineers.

[11] Lientz, B., & Swanson, E. (1980). *Software maintenance management: a study of the maintenance of computer application software in 487 data processing organisations.* Addison-Wesley.

[12] Oman, P., & Pfleeger, S. (1997). Applying software metrics. *IEEE Computer Society Press*.

[13] Pigoski, T. (1997). *Practice Software Maintenance: Best Practices for Managing Your Software Investment.* New York: John Wiley & Sons, Inc.

[14] Standish Group International. (1995). *CHAOS: Project failure and success report.* Retrieved August 1, 2010, from ProjectSmart: http://www.projectsmart.co.uk/docs/chaos-report.pdf

[15] Brooks, F. P. (1975). *The Mythical Man-Month.* Addison-Wesley.

[16] Moazeni, R., Link, D., Boehm, B., "Incremental Development Productivity Decline", presented at PROMISE 2013, Baltimore, MD, 2013