How CASE Automation Affects the Laws of Software Evolution:  
A Longitudinal Empirical Analysis

Evelyn J. Barry  
Texas A&M University  
College Station, TX  
ebarry@cgsb.tamu.edu  
(979) 845-2254 (phone)  
(979) 845-5653 (fax)

Chris F. Kemerer*  
University of Pittsburgh  
Pittsburgh, PA  
ckemerer@katz.pitt.edu  
(412) 648-1572 (phone)

Sandra A. Slaughter  
Carnegie Mellon University  
Pittsburgh, PA  
sandras@andrew.cmu.edu  
(412) 268-2308 (phone)  
(412) 268-7345 (fax)

April 25, 2005

* Contact author for this paper

Funded in part by National Science Foundation grants CCR-9988227 and CCR-9988315, a Research Proposal Award from the Center for Computational Analysis of Social and Organizational Systems, NSF IGERT at Carnegie Mellon University, the Sloan Software Industry Center at Carnegie Mellon University, and the Institute for Industrial Competitiveness at the University of Pittsburgh.
IEEE Keywords/Taxonomy:

D.2.2   Design Tools and Techniques
   D.2.2.a   CASE

D.2.7   Distribution, Maintenance, and Enhancement
   D.2.7.h   Maintenance management
   D.2.7.i   Maintenance measurement
   D.2.7.j   Maintenance planning

D.2.9   Management
   D.2.9.f   Planning
   D.2.9.g   Post-closure activities
   D.2.9.h   Productivity
   D.2.9.k   Project control & modeling
   D.2.9.l   Review and evaluation

D.2.18  Software Engineering Process
   D.2.18.a   Life cycle
   D.2.18.b   Process infrastructure
   D.2.18.c   Process measurement
   D.2.18.e   Software process models

D.2.8   Metrics/Measurement
   D.2.8.a   Complexity measures
   D.2.8.c   Process metrics

K.6.0   Management of Computing and Information Systems, General
   K.6.0.a   Economics

K.6.1   Project and People Management
   K.6.1.a   Life cycle
   K.6.1.b   Management techniques
   K.6.1.f   Systems development

K.6.3   Software Management
   K.6.3.a   Software development
   K.6.3.b   Software maintenance
How CASE Automation Affects the Laws of Software Evolution:  
A Longitudinal Empirical Analysis

Abstract

This research analyzes longitudinal empirical data on commercial software applications to test and better understand how software evolves over time, and to measure the long term effects of automated Computer Aided Software Engineering (CASE) tools on software productivity and quality. The research consists of two parts. First, we use data from source control systems, defect tracking systems, and archived project documentation to test Lehman’s laws of software evolution. We find empirical support for many of the laws, but not all. We then further analyze the data using moderated regression analysis to discern how CASE automation efforts at the research site influenced the software evolution lifecycles of the applications. Our results reveal that automation has enabled the organization to accomplish more work activities with greater productivity, thereby significantly increasing the functionality of the applications portfolio. And, despite the growth in software functionality, automation has helped to manage software complexity levels and to improve quality by reducing errors over time. Our models and their results demonstrate how longitudinal empirical software data can be leveraged to reveal the often elusive long term benefits of investments in software process improvement, and to help managers make more informed resource allocation decisions.
I. Introduction

Despite decades of experience the effective development of software remains a difficult challenge. Even after the introduction of a wide variety of process and technology innovations, numerous examples of failures in schedule, cost, and quality remain. Although there are no doubt myriad reasons for the continuing challenges in software development, one central problem is that it is difficult to distinguish the cause and effect relationships from implementing different development practices. In part, this is because the consequences from changes and innovations in software development practices are seldom immediate, but rather evolve over time. In addition, one could argue that the best way to study the impact of a particular software process innovation would be in a laboratory, where the software engineering researcher could control all aspects that are not of interest and could precisely link the practice innovation “treatment” to its effects. However, an experimental approach is less feasible when the effects emerge over a long period of time. As a result, it can be difficult to motivate the value of implementing new or modified practices today, when the intent is to improve software development performance on an ongoing basis for tomorrow. While the need to analyze software systems and the effects of development practice innovations over time has been recognized, the longitudinal data and the analytical approach needed to perform such analyses are often not available or are not able to be fully utilized.

The premise behind this research is that the longitudinal data that may be residing unanalyzed in software change logs and elsewhere are an extremely valuable resource that can be leveraged to address the challenge of determining the long-term impacts of changes in development practices. Organizations that systematically collect, organize, and report on data representing the state of their software systems have the opportunity to use these data to analyze trends and to discern the longer term effects of changes in software practices and procedures. Another goal of our research is to show how moderated regression analysis, which is frequently applied in the social sciences, can be
leveraged to isolate and understand the impacts of development practice innovations using longitudinal empirical software data.

Our research relies on analysis of detailed data from source code control systems, defect tracking systems and from archived project documentation. These data represent, in some cases, more than twenty years of software evolution at our dataset. As such they provide a relatively rare opportunity to investigate two central research questions. The first is: how do software systems evolve over time? While there has been some discussion and theorizing on this issue, there have been relatively few empirical studies to test these conjectures due to the difficulty in accessing longitudinal data. In particular, the software evolution “laws” originally proposed by Belady and Lehman can be evaluated using these data. As these laws are perhaps the earliest and most discussed of the research in the software evolution area, they are an appropriate starting point for this research [4, 30].

However, the second overall research question is designed to go beyond the general dynamics of systems changing over time due to entropy and related factors, and will focus on understanding the effects on software evolution of management-driven changes to the software development process, in this case, specifically, the automation of software development tasks through Computer Aided Software Engineering (CASE) tools. Due to the recognition that software development often consists of the systematic creation of components that must adhere to a well-specified set of constraints, the proposal to develop tools that would automate at least some of the required steps has been appealing from a variety of technical and economic perspectives [22, 23]. Automated development of software has the potential to reduce human error in the creation of code that must meet precise syntax and other constraints. It has the potential to produce similar or better software than that produced ‘by hand’ by relatively scarce skilled software development talent, potentially reducing costs. Automated development may lead to greater use of standardized components, thus
increasing software reliability and decreasing the future maintenance costs of software. Finally, automation may reduce the number of the less interesting, more mechanical tasks software developers have been required to do, thus freeing them to focus on tasks that require more creativity [22, 28]. On the other hand, some have questioned the extent to which automation can help software engineers to address the fundamental issues in software development such as complexity, reliability and productivity [7].

For all of these reasons software automation has been widely discussed, debated, critiqued or promoted. However, given that many of the proposed benefits of such automation tend to occur downstream and over the life cycle of the software systems, whereas the implementation and change costs tend to require significant investments in the current period, it has been difficult to demonstrate empirical evidence of the benefits of automation. However, this is exactly the kind of question for which longitudinal data could provide insight.

In response to this need for the analysis of long-term investments in software process improvement we have conducted an empirical evaluation of more than twenty years of software repository data from a commercial organization. Our analysis of this data begins with a test of Lehman's laws of software evolution to establish a benchmark. The results provide empirical support for many of the laws of software evolution defined by Lehman et al., but not for all. We then further analyze the data using moderated regression analysis to show how CASE automation efforts at the organization influenced the software evolution patterns over the complete lifecycles of the applications. Our results reveal that automation helped the organization to accomplish more work activities more productively, significantly increasing the functionality of the portfolio. At the same time, despite the growth in software functionality, automation helped manage software complexity levels and improved quality by reducing errors over time.
This paper is organized as follows. Section II describes some of the relevant prior research in this area with particular attention paid to the software evolution laws proposed by Lehman and his colleagues. Section III describes the first phase of the research where we develop models to test the laws of software evolution. Section IV then develops moderated regression models to analyze the impact of automated software development tools. We link the results of these two sections by showing how accounting for the impact of automation allows for a richer explanation of the software evolution phenomenon than is otherwise possible.

II. Prior Research
How are systems expected to behave over time? Although a tremendous amount of anecdotal evidence exists, there is relatively little carefully documented analysis due, in part, to the difficulty in collecting longitudinal data of this kind. Similarities in longitudinal behavior of software systems lie at the heart of the laws of software evolution. Challenges to empirical research on software evolution include differences in data collection at different sites, assembling and combining data from different studies and reconciling the characteristics of different studies and the interpretation of their results [33]. Given the large impact of software maintenance costs on information systems budgets, researchers and practitioners alike should prefer a scientific approach to examining the change processes in software systems. It will be difficult, if not impossible, to control lifecycle costs of software systems until software evolution can be better understood.

A. Laws of Software Evolution
The original and most well-documented attempt to study software evolution in a systematic way was conducted by Belady and Lehman beginning in the late 1960s [33]. Their early collaboration continued to expand over the next decade [4, 5, 33], and resulted in a set of “laws” of software evolution [4, 6]. In a seminal paper Belady and Lehman outline three laws of software evolution: (i) the law of continuous change, (ii) the law of increasing entropy, and (iii) the law of statistically
smooth growth. The proposed laws were based on empirical studies of the OS/360 operating system through 21 versions released by IBM [4]. In a later paper Lehman revised the initial three laws and renamed them: (i) the law of continuing change, (ii) the law of increasing complexity (formerly the law of increasing entropy), and (iii) the law of self regulation (formerly the law of statistically smooth growth). In addition, he added two new laws, the law of conservation of organizational stability (aka invariant work rate) and the law of conservation of familiarity [25]. These two additional laws describe limitations on software system growth.

Lehman’s research found that once a module grew beyond a particular size such growth was accompanied by a growth in complexity and an increase in the probability of errors [27]. By the late 1990s, three additional laws of software evolution had been proposed: the law of continuing growth, the law of declining quality and the feedback system [30]. He presents the feedback system law in two assertions. Assertion 1 states: “The software evolution process for E-type systems¹, which includes both software development and its maintenance, constitutes a complex feedback learning system.” Assertion 2 states: “The feedback nature of the evolution process explains, at least in part, the failure of forward path innovations such as those introduced over the last decades to produce impact at the global process level of the order of magnitude anticipated.” [29].

In Table 1 we summarize the current laws using their most current names and definitions, and order them by three broad categories: (i) laws about the evolution of software system characteristics; (ii) laws referring to organizational or economic constraints on software evolution; and (iii) “meta-laws” of software evolution².

¹ Lehman and his colleagues often reference “E-type systems” with respect to the laws of software evolution, those systems that are “developed to solve a problem or implement an application in some real world domain.”[3] As all of the systems discussed and analyzed here are of this type, we have eliminated this excess stipulation in the discussion that follows to simplify the narrative for the reader.

² Over the course of the research in this area some laws have been added and some have been renamed. The change in the number of laws, in particular, makes referencing them by number potentially confusing. Therefore, we have
### Laws of Software Evolution

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Systems must continually adapt to the environment to maintain satisfactory performance</td>
<td>Functional content of systems must be continually increased to maintain user satisfaction</td>
<td>As systems evolve they become more complex unless work is specifically done to prevent this breakdown in structure</td>
<td>System quality declines unless it is actively maintained and adapted to environmental changes</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Description</th>
<th>Law of Conservation of Familiarity</th>
<th>Law of Conservation of Organizational Stability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incremental rate of growth in system size is constant to conserve the organization’s familiarity with the software.</td>
<td>The organization’s average effective global activity rate is invariant throughout system’s lifetime</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Description</th>
<th>Law of Self Regulation</th>
<th>Law of System Feedback</th>
</tr>
</thead>
<tbody>
<tr>
<td>The software evolution processes are self-regulating and promote globally smooth growth of an organization’s software</td>
<td>Software evolutionary processes must be recognized as multi-level, multi-loop, multi-agent feedback systems in order to achieve system improvement</td>
<td></td>
</tr>
</tbody>
</table>

---

### Table 1: Laws of Software Evolution [30]

#### B. Prior Empirical Validation Studies

A variety of authors have attempted empirical tests involving the laws. In their original studies of software evolution, Belady and Lehman analyzed observations of 21 releases of OS/360, an operating system for large mainframe computers. They used the system size (module count) and the software release sequence number to evaluate the laws of continuing change and increasing complexity. A number of studies have used the same measures on other software systems to evaluate the same group of laws, and have employed least squares linear regression and inverse squares in the analysis [4, 25, 30].

To test the laws of conservation of organizational stability and familiarity Lehman ran regressions using the change in number of modules as the dependent variable [26]. Results confirmed that the organization performing software maintenance displayed an invariant work rate and conservation of familiarity [25].

---

adopted the convention in the paper of referring to the laws by name and, in particular, after this review of prior literature we will use only the most modern name for each law.
In later studies, Chong Hok Yuen, a student of Lehman’s, conducted a study on 19 months of ongoing maintenance data for a large software system. He was able to collect the ‘bug reports’ and ‘bug responses’ documenting the number of modules handled for each report, as well as the total number of modules in the software system for each ‘bug report’ [9]. Analyzing size (in modules), cumulative modules handled, and fraction of modules handled, the research provided empirical support for the laws of continuing change, increasing complexity and continuing growth. The data and analysis failed to support the law of declining quality [10, 11].

Cooke and Roesch [13] analyzed data from 10 releases of a real-time telephone switching software system. The data were collected for 18 months of software modification. Their work supported the laws of continuous change, increasing complexity, and continuing growth. Their work failed to support the law of conservation of organizational stability.

Lehman, Ramil, et al. presented a test of six laws in a 1997 conference paper. Analyzing data from software modifications to a financial transaction system, they were able to test and support five of the eight laws of software evolution: continuous change, increasing complexity, continual growth, conservation of organizational stability, and feedback system. [30]

That same year Gall, et al. published a conference paper that presented data plots from multiple releases of a telecommunications switching system [17]. They argue that their plots show support for the laws of continuous change and continuous growth. However, the authors note that there are subsystems that appear to exhibit a completely different behavior.

The following year Lehman, et al. presented a new paper from their FEAST (Feedback, Evolution, And Software Technology) project which is based on empirical data from ICL’s VME operating system kernel, Logica’s FW banking transaction system and a Lucent Technologies real time system [31]. They also find support for the laws of continuous change and continuous growth in
functionality, but note the difficulty in collecting appropriate data to test the laws concerning software complexity, quality, organizational work-rate and software process feedback systems.

More recently, Burd, *et al.* use a reverse engineering approach to track cumulative changes in call and data dependencies across versions of software. They argue that their data support the law of feedback systems [8].

The Law of Conservation of Familiarity states that the incremental growth rate is constant. A few studies have not supported this statement. Empirical work examining open source software includes research by Godfrey and Tu who track growth in lines of code (LOC) from Linux and note that it is ‘super linear’\(^3\), and does not remain constant over time [18]. Aoki *et al.* use data from 360 versions of the open source system JUN and find that the growth of their system is also at a ‘super linear’ rate, and because the growth is not constant, the law of conservation of familiarity is not supported [2]. In each of these studies the time series equates a release sequence number with one unit of time\(^4\).

Finally, a 2004 paper by Paulson *et al.* combines both the method used by Gall *et al.* and the open source orientation of Godfrey *et al.* and Aoki, *et al.* [42]. Changes in software releases were plotted on a time interval of number of days. Software size was recorded on the date of release, not release sequence number. In comparing the growth rates of SLOC in both open and closed source systems Paulson *et al.* found the rates to be similar, thus suggesting support for the feedback law [42].

### C. Summary of Prior Research

From this summary of prior research a few things seem clear. The first is that the laws of software evolution have merited attention from a variety of researchers. Understanding the behavior of

---

\(^3\) ‘super linear’ is interpreted to mean an increasing non-linear curve.

\(^4\) Note, however, that in this paper the results are presented as a concave curve when size is graphed versus release. The x-axis is labeled with the release dates, using a constant interval. However, it is not clear how this relates to the data, whose actual release dates do not appear to be at constant intervals. This suggests the importance of using actual dates, rather than release numbers as proxies, when the actual dates are available in the dataset.
software systems over time is generally seen as a worthy research goal, and the laws, which had their origins in the late 1970s, continue to be the extant model on the subject today. However, from an empirical point of view, support for the laws has been mixed. Researchers are generally unable to test more than a small number of the laws, and even then, data limitations tend to severely constrain the analysis.

Given this prior research, starting the analysis of our empirical data with an evaluation of the laws of software evolution provides a clear benchmark against the prior and current literature. Therefore, the first phase of the analysis will be to subject the laws of software evolution to their most comprehensive independent test to date. This will then form a baseline for using the longitudinal data to assess the impact of software automation tools on how the organization’s software has evolved over time.

III. Research Model – Phase One

A. Longitudinal Data Set

The ‘candidate system’ in this work is a portfolio of application software systems for a large retail company. The company had a centralized IT group that was responsible for developing and supporting 23 application systems. The applications cover four broad domains, i.e. human resources, finance, operations and merchandising. The Laws of Software were initially developed through empirical observations of the IBM OS/360 operating system. An operating system contains a large number of subsystems that may individually be updated or enhanced independently. Many of the subsystems might remain untouched for long periods of time, while some will be executed constantly. We analyze this software portfolio in much the same way. The applications included in the portfolio are all relevant to the retailer. Some of the applications will be used daily, while others

5 The specific applications include Advertising, Accounts Payable, Accounts Receivable (3 applications), Sales Analysis (2 applications), Capital Pricing Management, Fixed Asset Management, Financial Reporting, General Ledger, Shipping, Merchandising (3 applications), Order Processing, Pricing (5 applications), Payroll, and Inventory Management.
may be relatively dormant. The applications in our software portfolio are all related to supporting the information requirements for the retailer, in much the same way the subsystems in an operating system are relevant to successful operation of an operating system.

Most importantly for the research, for more than twenty years the IT group maintained a log of each software module created or modified in the portfolio. All software modifications were recorded as log entries at the beginning of each software module [24]. The coded logs yield a record of 28,000 software modifications to almost four thousand software modules. This is a rich dataset that affords a rare opportunity to observe software evolve over a 20-year time period.

Of course, as it is a longitudinal dataset of considerable longevity the underlying technological artifacts were written and maintained with contemporaneous technologies, i.e. programming languages and tools that would not be at the cutting edge today, e.g., COBOL. Obviously today’s more recent technology choices do not have a twenty year history, and therefore in order to study long term change the focus is appropriately on higher level, more abstract phenomenon, rather than a more narrow focus on optimizing specific technologies. Thus, in this research the focus is on the broad effects of CASE automation on managerially-relevant dimensions such as productivity and quality.

We supplemented the detailed change log entries with archival records obtained from the research site’s source code library to capture basic software product measures of size and module age. Each software module was analyzed using a code complexity analysis tool to yield measures of software complexity. In addition, the source code library helped us to identify the use of automation in software development at the research site by indicating which modules were developed and/or maintained using the automated software tool.
Using the encoded maintenance log we constructed a time series data panel to describe the number and types of lifecycle maintenance activities for all applications for each month in the software application portfolio’s life span. Module level data were aggregated each month to compute portfolio-level metrics for size and complexity. One advantage of this dataset is the use of actual date data, as opposed to being limited by the data to using the proxy of release number as has been done in some earlier work.

Table 2 provides the summary statistics for the variables used in our analysis. The statistics reflect the values of the measures computed for each month.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Measurement</th>
<th>Mean</th>
<th>Std. Dev.</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Age</td>
<td>Age of the portfolio in months</td>
<td>122.5</td>
<td>70.6</td>
<td>1.0</td>
<td>244.0</td>
</tr>
<tr>
<td>Age-squared</td>
<td>Age in months * Age in months</td>
<td>4961.3</td>
<td>4446.5</td>
<td>0.3</td>
<td>14762.3</td>
</tr>
<tr>
<td>No. of activities</td>
<td>Count of corrections, adaptations, enhancements and new module creations to the portfolio</td>
<td>107.3</td>
<td>132.0</td>
<td>0.0</td>
<td>568.0</td>
</tr>
<tr>
<td>Module count</td>
<td>Count of number of modules in the portfolio</td>
<td>1029.2</td>
<td>1192.4</td>
<td>5.0</td>
<td>3609.0</td>
</tr>
<tr>
<td>Cyclomatics per module</td>
<td>Total cyclomatic complexity of the modules in the portfolio divided by number of modules</td>
<td>108.4</td>
<td>18.0</td>
<td>48.3</td>
<td>161.8</td>
</tr>
<tr>
<td>Operands per module</td>
<td>Total operands in the portfolio divided by number of modules</td>
<td>1384.0</td>
<td>306.2</td>
<td>570.9</td>
<td>1910.5</td>
</tr>
<tr>
<td>Calls per module</td>
<td>Total calls in the portfolio divided by number of modules</td>
<td>17.4</td>
<td>11.3</td>
<td>4.4</td>
<td>38.2</td>
</tr>
<tr>
<td>No. of corrections per module</td>
<td>Total corrections divided by number of modules</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.1</td>
</tr>
<tr>
<td>Percentage growth in module count</td>
<td>Change in number of modules this month divided by the total number of modules last time period</td>
<td>0.0</td>
<td>0.1</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>No. of activities per developer</td>
<td>Count of corrections, adaptations, enhancements and new module creations divided by the count of developers making modifications during the month</td>
<td>4.8</td>
<td>3.3</td>
<td>1.0</td>
<td>17.2</td>
</tr>
<tr>
<td>No. of developers</td>
<td>The number of developers working on the portfolio that month</td>
<td>15.9</td>
<td>16.3</td>
<td>0.0</td>
<td>53.0</td>
</tr>
</tbody>
</table>

Table 2: Descriptive Statistics

Our data panel consists of 244 monthly observations. To analyze longitudinal data of this sort we use a time-series regression in evaluating each of the laws of software evolution. As is common with many datasets for time-series analyses, we found serial correlation [19]. Thus, we used the
Prais-Winsten estimators to correct for serial correlation, using the AR1 correction [19]. The results reported in the following analyses have all used the Prais-Winsten estimators as implemented in Stata.

B. Phase One Modeling

The basic argument behind the laws of software evolution is that software systems change over time. Therefore, the base case version of our model uses a single variable, system age, to test the laws.

The general form of the model is:

\[ Y_{Lt} = \alpha_L + \beta_L \cdot AGE_t + \epsilon_{Lt} \]

where \( Y_t \) represents the particular dependent variable used to evaluate each law \( L \) for time period (month) \( t \), \( AGE \) is the variable for system age that varies by time period (month) \( t \), and \( L \) ranges from one to six to represent each of the laws evaluated.\(^7\) [19, 43]

Table 3 provides a summary of the estimation for each of the laws of software evolution.

1. Law of continuous change

The first law, the law of continuous change, states that “[a] system must be continually adapted else it becomes progressively less satisfactory in use” [32]. For the dependent variable we use a count of all changes and additions to the software portfolio. Our results reveal that the coefficient on the AGE variable is positive and significant, supporting the law of continuous change, and suggesting that the number of maintenance activities performed averages 107 each month and increases with age at a rate of more than 1.4 additional activities each month. AGE explains a significant proportion (about 23%) of the total variation in the evolution of software portfolio activities.

\(^6\) Stata version 8 available from Stata Corporation.

\(^7\) Note that this model is originates from the theory proposed by Lehman, et al., and is not generated from the data set being analyzed. Therefore, and consistent with standard econometric approaches, it is not appropriate to create sub-sets of the data, e.g., “holdout samples” as is appropriate in other, data-driven modeling work [19, 43].
<table>
<thead>
<tr>
<th>Law</th>
<th>Dependent variable</th>
<th>Adjusted $R^2$</th>
<th>$\beta_L$ (std error)</th>
<th>Confirmed?*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Continuous change</td>
<td>No. of activities</td>
<td>0.2294</td>
<td>1.4312*** (0.1656)</td>
<td>Yes</td>
</tr>
<tr>
<td>Continuous growth in</td>
<td>Module count</td>
<td>0.0126</td>
<td>14.8314*** (1.1346)</td>
<td>Yes</td>
</tr>
<tr>
<td>functionality</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Increasing complexity</td>
<td>Cyclomatics per</td>
<td>0.1744</td>
<td>-0.0961 (0.1072)</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>module</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Operands per</td>
<td>0.0895</td>
<td>-0.3838 (1.8121)</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>module</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Calls per module</td>
<td>0.0100</td>
<td>0.0158 (0.0447)</td>
<td>No</td>
</tr>
<tr>
<td>Declining quality</td>
<td>No. of corrections per module</td>
<td>0.0322</td>
<td>0.00006*** (0.0000)</td>
<td>Yes</td>
</tr>
<tr>
<td>Conservation of familiarity</td>
<td>Percentage growth in module count</td>
<td>0.0212</td>
<td>-0.0002* (0.0001)</td>
<td>Yes</td>
</tr>
<tr>
<td>Conservation of</td>
<td>No. of activities per developer</td>
<td>0.1902</td>
<td>0.0334*** (0.0039)</td>
<td>No</td>
</tr>
<tr>
<td>organizational stability</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notes: $n = 244; \dagger p < 0.10; * p < 0.05; ** p < 0.01; *** p < 0.001.

Table 3: Summary of Phase One Results

2. Law of continuous growth

The closely related law, continuous growth in functionality, is also supported. This law is tested using the cumulative number of modules as the dependent variable. AGE again is highly significant as an explanatory variable, although the overall variation explained is much less than for the first law. AGE explains about 1% of the total variation in growth in module count. The cumulative number of modules grows at a rate of about 15 per month.

3. Law of increasing complexity

The next law in the subgroup of laws relating to software system characteristics is the law of increasing complexity. The law does not specify a particular complexity metric; for the dependent variable we use the McCabe Cyclomatic complexity per module [36]. However, we also examine two other measures of software complexity: operands per module [20] and calls per module. Although still in use, these metrics were specifically chosen as they represent contemporaneous metrics for the software examined. And, we use multiple metrics to guard against the possibility that any results are somehow metric-specific. Despite this, we did not find empirical support for this law for any of the three different measures of software complexity since the estimated
coefficients on the AGE variable are not significantly different from zero at the usual statistical levels. However, it is important to note that the law contains the caveat that increasing complexity is expected unless steps are taken to mitigate it. We defer further analysis of these results and this question until the second part of our modeling documented below as Phase Two.

4. Law of declining quality
The fourth and final of the software system characteristic laws is the law of declining quality. In this analysis we use the corrections per module as the dependent variable. The results of this analysis provide support for this law, as the coefficient on AGE is positive and significant. However, the estimated coefficient value is almost zero (0.0000623), suggesting that the corrections per module increase only very slightly with age, i.e., are almost constant over time. Further, AGE explains only about 3% of the total variation in the number of corrections per module.

5. Law of conservation of familiarity
The next major group of laws is that relating to Operational or Economic Resource constraints. The law of conservation of familiarity is tested using the percentage growth in the number of modules each month. Note that this is different from the test of the earlier law above where the actual number of modules was the dependent variable. Here, the model seeks to explain variation in the rate of growth in the number of modules in the portfolio – that is, does the number of modules added as a percentage of the total number of modules in the portfolio change at a constant, declining or increasing rate? The statistical result for this analysis is that the coefficient on AGE is negative and significant, which is interpreted as providing support for the law, as the percentage growth in modules does not increase over time. In particular, in the model AGE explains only about 2% of the variation in percentage growth of modules over time, and the coefficient on AGE is very small (close to zero), implying that the decrease in the percentage growth rate with AGE is very small, i.e., that the actual percentage growth is nearly constant over time.

The law of conservation of organizational stability states that the amount of work accomplished per developer will be constant over time. The dependent variable in this analysis is the count of all changes and additions made per developer. Our results do not support this law as the number of activities per developer actually significantly increases with age. In fact, the average number of activities per developer is almost five per month, and this number increases at a rate of 0.03 additional activities per month. AGE explains almost 20% of the total variation in work rate. Of course, this result immediately raises the question of why productivity is increasing over time, a question we will explore further in Phase Two of the analysis.

7. The ‘meta-laws’

Finally, two of the laws of software evolution can actually be seen as ‘meta-laws’. The law of self regulation states that “global E-type system evolution processes are self-regulating” [32]. The other ‘meta-law’ is known as the feedback system which states that “evolution processes are multi-level, multi-loop, multi-agent feedback systems” [32]. These laws describe the relationships between software systems and the organizational and economic environments in which those systems exist. At the abstract level of description of these laws it is difficult to say what empirical model could be formally tested to support or reject these laws. However, overall, the results for the first six laws do suggest general support for these laws – we find that although the portfolio is growing in size over time, the level of complexity is not increasing, and the rate of growth is constant. Further, developers are accomplishing more work and significantly increasing the functionality of the portfolio, but despite this, the quality of code does not decline significantly – all of this suggests that processes of self-regulation and feedback are operating – evolution seems to be happening at a very

---

8 Note that although the estimated coefficient is highlighted as statistically significant, it is statistically significant in the opposite direction predicted by the law.
“controllable” pace, without significantly increasing or decreasing the complexity and quality of the portfolio.

C. Extension of Phase One Analysis
One extension of these results is to estimate a slightly more sophisticated quadratic model of the form:

\[ Y_{lt} = \alpha_t + \beta_{L1} \cdot AGE_{lt} + \beta_{L2} \cdot AGE_{lt}^2 + \epsilon_{lt} \]

where the variables are the same as in the prior model with the addition of an \( AGE^2 \) term which is added to allow for a non-linear relationship in the software evolution pattern exhibited over time.

Recent research has hypothesized that some systems display a ‘super-linear’ growth [2, 18, 42]. For example, these research projects and others have proposed that growth can be accelerated if the software system is an open-source system or if extreme programming has been used to write or maintain the system. The empirical results appear to be mixed. Aoki, et al. found super-linear growth in the releases of JUN [2], while others have found that software grows at a linear rate when source code growth rate is measured over elapsed time rather than the growth between release sequence numbers [42].

Given these mixed results we extended our model to check the quadratic form to see how our estimated regressions will compare with the results for our linear specification of the equations for each law. The quadratic form of the model allows for both a linear and a non-linear effect of AGE on the dependent variable.

Using the same set of dependent variables as in the first phase of the analysis, we estimated the new model and present the results in Table 4. This further analysis does not change any of the main results of the earlier section, as laws that were supported in the simpler, linear formulation continue to be supported with the non-linear model, and vice versa. Adding the quadratic term increases the
variation explained in the dependent variable (in terms of adjusted R-squared) by significant amounts for the first two laws tested, but has relatively little effect on the tests of the other laws. This suggests that system change and growth over time may be more accurately modeled as a non-linear function, as fit improves for the two original laws and does not appreciably decline for the others.

<table>
<thead>
<tr>
<th>Law</th>
<th>Dependent variable</th>
<th>Adjusted $R^2$</th>
<th>$\beta_{L_1}$ (std. error)</th>
<th>$\beta_{L_2}$ (std. error)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Continuous change</td>
<td>No. of activities</td>
<td>0.6356</td>
<td>1.4708*** (0.0724)</td>
<td>0.0042*** (0.1579)</td>
</tr>
<tr>
<td>Continuous growth in functionality</td>
<td>Module count</td>
<td>0.2976</td>
<td>169.2190† (95.4203)</td>
<td>0.2385* (0.0985)</td>
</tr>
<tr>
<td>Increasing complexity</td>
<td>Cyclomatics per module</td>
<td>0.1559</td>
<td>-0.1131 (0.1218)</td>
<td>0.0026† (0.0014)</td>
</tr>
<tr>
<td></td>
<td>Operands per module</td>
<td>0.1031</td>
<td>-0.3261 (1.8121)</td>
<td>0.0326† (0.0187)</td>
</tr>
<tr>
<td></td>
<td>Calls per module</td>
<td>0.0243</td>
<td>0.0166 (0.0431)</td>
<td>0.0007* (0.0004)</td>
</tr>
<tr>
<td>Declining quality</td>
<td>No. of corrections per module</td>
<td>0.0420</td>
<td>0.0001** (0.0000)</td>
<td>-5.51e-07† (3.09e-07)</td>
</tr>
<tr>
<td>Conservation of familiarity</td>
<td>Percentage growth in module count</td>
<td>0.0190</td>
<td>-0.0002** (0.0001)</td>
<td>8.77e-07 (1.37e-06)</td>
</tr>
<tr>
<td>Conservation of organizational stability</td>
<td>No. of activities per developer</td>
<td>0.1944</td>
<td>0.0329*** (0.0039)</td>
<td>0.0001 (0.0001)</td>
</tr>
</tbody>
</table>

Notes: $n = 244$; † $p < 0.10$; * $p < 0.05$; ** $p < 0.01$; *** $p < 0.001$.

Table 4: Phase One results – Non-linear model

IV. Research Model – Phase Two

In Phase One of this analysis we applied a basic test of software evolution by using AGE of the system as a predictive variable to test a variety of hypotheses about how software evolves. This type of analysis can be useful in informing software developers and managers about the level of change and growth in their software systems that may be expected over time. However, the results from Phase One are unable to offer very much in the way of insights into the process behind software evolution and, in particular, what effect managerial actions might have on the evolution patterns of software systems. Use of a detailed, longitudinal dataset and a moderated regression analysis can help to determine the causes of changes, especially where developers and managers are continually trying to improve software processes.
A. Prior research on software automation evaluation

Early in its history software automation acquired the label of “CASE Tools” – Computer Aided Software Engineering Tools. Although there is a large literature in economics on the general effects of automation on production processes, software engineering automation has its own specialty, given that the automation seeks to enhance the mental, rather than the physical, attributes of the worker. Software developers are a special form of knowledge worker since they are highly computer literate and, all else being equal, could be expected to master the use of a computer-based automation tool better than perhaps any other professional. Finally, the great need for computer software, prompted in part because of the dramatic decline in computer hardware costs, created a tremendous demand for software development labor. Just as classic microeconomic theory would predict, increased demand for labor drove its price higher and made the notion of substituting capital, in the form of automation tools, for labor increasingly attractive. This was coupled with the notion that a computer-based tool had the capacity to actually improve the quality of the resulting software product, given the tool’s ability to perform repeated functions without the kind of errors that ‘hand-crafting’ software inevitably produces [35].

These kinds of needs and analyses produced a range of tool solutions, some focusing on assisting the upfront activities of analysis and design and others directly automating the production of computer code. However, despite great interest and rapid sales of tools, early results on their effects were often disappointing compared to the initial expectations. Preventing greater understanding of this phenomenon, however, was the near complete lack of quantitative data on the impact of tools, particularly their long-term effects. Most empirical studies relied on subjective assessments of managers and developers involved in software projects to report their perceptions of the impact [16, 21, 34, 37, 38].
Some early work on quantitative measures that appeared in *IEEE Transactions on Software Engineering* was promising for automated software tool usage, e.g. with respect to aiding software reuse [3]. However, a significant fraction of implementations reported difficulties in gaining wide-scale adoption of the automation tools. This was due, in particular, to the difficulty in providing evidence of meeting *a priori* expectations for automation benefits. Typical of this type of concern and, ultimately, a barrier to adoption was a contemporaneous report from 1989 that noted:

"...many people don't believe that [the tool] saves money....The consensus is that unless [it] delivers the productivity benefits, the support from above will disappear."

[39] p. 36

A variety of practitioner-oriented articles acknowledged and reported on the difficulties organizations were having in making a business case for the investment in software automation [14, 44]. Academic research also focused on the technology adoption problem found when trying to implement these tools in a software development organization. Researchers approached the problem from a variety of viewpoints, including economics [23] and organizational behavior [41]. And the focus of other research shifted to the failure of organizations to successfully adopt these tools [21]. Research looking at adoption in software process innovations showed that, even when successful, adoption took a much longer time than was previously believed [15].

How does the innovation of software automation fit into the view of software evolution? One starting point can be found in the notions of Lehman himself, in a 1990 conference paper [28]. In it Lehman argues that the fundamental challenge in managing software development is managing uncertainty. He notes that automation tools have a “special role in reducing and controlling ambient uncertainty in system behavior.” p. 240 [28]. More specifically, and with an eye toward the ongoing debate about the justification of tool usage, he states that,

“The visible benefits derived from such activity and from commercial...products arising therefrom [sic] are, however, not widely perceived. In part this is due to the fact that many of the benefits are anti-regressive...They lead to a reduction in the
He also argues, as do many authors, that automated tools aid the software development process by transferring the “repetitive, mechanical, aspects of software development to computers, leaving people to concentrate more and more on the judgmental, creative, decision making activities within and in support of the process.” p. 243 [28] Finally, and not surprisingly, since the underlying economic conditions that produced the interest in, and demand for, software automation tools remain relatively unchanged today, there continues to be current interest in tool usage and its impact on performance, e.g., the 2004 article by Limayem, et al. [34]. And, this interest can be expected to persist, especially as organizations continue to look for ways to control and reduce software labor costs, e.g., the current offshore outsourcing phenomenon. The software repository data at this organization affords us the opportunity to investigate the actual impact of these tools over time, in contrast to most of the literature described above, which was limited to measures of participants’ perceptions of the impact of the tools.

B. Automation at the data site
The IT group at our research site introduced automation tools approximately half-way through our data collection period (month 125 of 244). Their choice was the implementation of a source code generation tool\(^9\). The company had gone through a series of mergers and acquisitions that resulted in a dramatically changed set of application software systems. The tools were intended to help with the expanded responsibilities of the centralized IT group, in terms of improving developer productivity and reducing software development and maintenance costs. Limayem, et al. note that “The greatest benefit of…tools lies in their capability of generating computer module codes. This

\(^9\) Brand name Telon, a set of productivity tools available from Computer Associates. CA-Telon has a number of tools for several different platforms. During the longitudinal data collection period the research site used the code generator for COBOL mainframe applications.
capability can greatly improve developers’ productivity as well as the quality, reusability, portability, and scalability of applications.” p. 154 [34].

C. Data on Automation

The modules developed and maintained with the tool were marked in the archival data for the software portfolio as to whether or not they were automatically generated. In our time-series data the variable TOOL is calculated as the number of tool-generated modules divided by the total number of modules in the portfolio. Thus, the variable TOOL, is the proportion of application portfolio in month t that has been created or maintained using CASE automation.

Figure 1 illustrates the growth of the portfolio in terms of numbers of modules. When the tools are introduced at month 125 the portfolio is growing very rapidly. Although the percentage of the portfolio created using automation tools peaks within the first two years of their introduction, more than one-quarter of all modules are developed with tool support.

![Portfolio Growth and the Growth of Modules Developed using Automation tools](image)

**Figure 1: Portfolio Growth and the Growth of Modules Developed using Automation tools**

Table 5 provides the descriptive statistics comparing the modules developed via the automated tool or manually. From Table 5 it can be casually observed that software modules developed with the aid of automation tools tend to be larger, and more complex on a variety of dimensions, all else
being equal. In the next section we will statistically test for the effect of automation on the evolution of the software portfolio.

<table>
<thead>
<tr>
<th>Module dimension</th>
<th>Developed with Automation Tools (n= 956 modules)</th>
<th>Developed Manually (n= 2653 modules)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>Std. dev.</td>
</tr>
<tr>
<td>Lines of code</td>
<td>5840.3</td>
<td>497.4</td>
</tr>
<tr>
<td>Total cyclomatics</td>
<td>239.2</td>
<td>33.7</td>
</tr>
<tr>
<td>Total operands</td>
<td>3474.8</td>
<td>488.2</td>
</tr>
<tr>
<td>Total calls</td>
<td>85.4</td>
<td>11.1</td>
</tr>
</tbody>
</table>

Table 5: Descriptive Statistics for the Software Module Characteristics as a Function of Automation

D. Phase Two Modeling and Results

To evaluate the impact of automation on the evolution of the software portfolio, we adopt a moderated regression approach [1, 12]. As we noted earlier, moderated regression analysis is frequently used in the social sciences as well as in business, education, communication, and other disciplines to investigate the interaction of two or more independent variables. In moderated regression the researcher is attempting to understand more than the linear and additive effects of some predictor variables on some outcome variable. Rather, in addition to these effects, the researcher wants to understand whether the value of an outcome variable depends jointly upon the value of the predictor variables\(^{10}\).

In our study we use moderated regression to understand how the use of automation influences the way in which the software portfolio evolves. We extend our previous model to include variables representing the extent to which automation tools were used, and the associated interaction variables associating TOOL use with AGE and AGE\(^2\). With this extended model we can investigate the effect TOOL use may have on software evolution. For example, the use of automation could significantly increase the average number of modules of the portfolio (this is shown if the

---

\(^{10}\) For example, a study of workspace design investigated the turnover of personnel in the office, examining predictor variables including the number of people in the office and the design of each individual’s workspace [40] Oldham, G. and Y. Fried, "Employee Reactions to Workplace Characteristics," Journal of Applied Psychology, vol. 72, pp. 75-80, 1987. Those researchers used moderated regression to perform the analysis and found a significant interaction between the predictors such that the greater the number of individuals in the office, the higher the rate of personnel turnover. However, the relationship between number of individuals and turnover was weakened when individuals’ desks were enclosed by partitions.
coefficient on the TOOL variable is positive and significant). We can also look at the interaction effects to see if automation either amplifies or counteracts the effects of AGE on software characteristics. For example, automation could significantly increase the rate of growth of modules (this would be indicated if the coefficient on the interaction of \(AGE \times TOOL\) were positive and significant). Or, automation could be seen as causing the portfolio to grow at an increasing or decreasing (non-linear) rate (this could be observed if the coefficient on the \(AGE^2 \times TOOL\) interaction were significant). Table 6 shows the descriptive statistics for the TOOL automation variable and the associated interaction terms.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Mean</th>
<th>Std. Dev.</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOOL</td>
<td>0.1</td>
<td>0.1</td>
<td>0.0</td>
<td>0.4</td>
</tr>
<tr>
<td>TOOL*AGE</td>
<td>8.2</td>
<td>4.5</td>
<td>-0.9</td>
<td>16.6</td>
</tr>
<tr>
<td>TOOL*AGE^2</td>
<td>20.6</td>
<td>870.7</td>
<td>-1896.2</td>
<td>2014.2</td>
</tr>
</tbody>
</table>

Table 6: Descriptive statistics for the TOOL automation variables

We estimate the following moderated regression model to evaluate the impact of automation on software evolution profiles:

\[ Y_{lt} = \alpha_l + \beta_{l1} \times AGE_t + \beta_{l2} \times AGE_t^2 + \beta_{l3} \times TOOL_t + \beta_{l4} \times TOOL_t \times AGE_t + \beta_{l5} \times TOOL_t \times AGE_t^2 + \epsilon_{lt} \]

where \(Y_{lt}\) represents the particular dependent variable evaluated for each law estimated at time period (month) \(t\), and “L” can range from one to six to represent each of the six laws. Table 7 presents the results of validating the laws of software evolution taking into account the effect of software automation tools.
<table>
<thead>
<tr>
<th>Law</th>
<th>Dependent variable</th>
<th>Adj. R²</th>
<th>(\beta_{L1})</th>
<th>(\beta_{L2})</th>
<th>(\beta_{L3})</th>
<th>(\beta_{L4})</th>
<th>(\beta_{L5})</th>
</tr>
</thead>
<tbody>
<tr>
<td>Continuous change</td>
<td>No. of activities</td>
<td>0.7651</td>
<td>2.715*** (0.0805)</td>
<td>-0.0115*** (0.0057)</td>
<td>127.3557† (162.3078)</td>
<td>16.1686*** (5.9278)</td>
<td>-0.1268*** (0.0493)</td>
</tr>
<tr>
<td>Continuous growth in</td>
<td>Module count</td>
<td>0.5534</td>
<td>19.9264*** (1.6372)</td>
<td>0.0546*** (0.0096)</td>
<td>-285.5146* (187.8702)</td>
<td>32.717*** (9.5351)</td>
<td>-0.3068*** (0.0865)</td>
</tr>
<tr>
<td>functionality</td>
<td>Cyclomatics per</td>
<td>0.2945</td>
<td>0.015703 (0.3252)</td>
<td>0.0017 (0.0023)</td>
<td>134.724** (51.9728)</td>
<td>0.6451 (2.4565)</td>
<td>-0.0195 (0.0199)</td>
</tr>
<tr>
<td></td>
<td>module count</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Operands per</td>
<td>0.3556</td>
<td>2.057469 (3.6104)</td>
<td>216.02*** (0.0255)</td>
<td>10.3776 (575.5422)</td>
<td>-0.3094 (27.2255)</td>
<td>1236.38*** (0.2209)</td>
</tr>
<tr>
<td></td>
<td>module</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Calls per module</td>
<td>0.6730</td>
<td>0.0787† (0.0506)</td>
<td>0.0002 (0.0003)</td>
<td>73.485*** (6.4603)</td>
<td>0.5842* (0.3237)</td>
<td>-0.00891** (0.0029)</td>
</tr>
<tr>
<td>Increasing complexity</td>
<td>No. of corrections</td>
<td>0.3549</td>
<td>-0.0002* (0.0001)</td>
<td>5.98e-07 (5.90e-07)</td>
<td>0.1047*** (0.0167)</td>
<td>-0.0013* (0.0006)</td>
<td>5.18e-06 (5.90e-06)</td>
</tr>
<tr>
<td>Declining quality</td>
<td>per module</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conservation of familiarity</td>
<td>Percentage growth</td>
<td>0.0200</td>
<td>-0.0005 (0.0007)</td>
<td>5.04-e-06 (5.11-e-06)</td>
<td>0.1690 (0.1444)</td>
<td>-0.0045 (0.0053)</td>
<td>5.26e-07 (0.0000)</td>
</tr>
<tr>
<td></td>
<td>in module count</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conservation of</td>
<td>No. of activities</td>
<td>0.4448</td>
<td>0.0123 (0.0217)</td>
<td>0.0001 (0.0002)</td>
<td>13.703*** (4.2443)</td>
<td>-0.0177 (0.1569)</td>
<td>-0.0002 (0.0014)</td>
</tr>
<tr>
<td>organizational stability</td>
<td>per developer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notes: n = 244; † p < 0.10; * p < 0.05; ** p < 0.01; *** p < 0.001.

Table 7: Results of Phase Two Analysis

1. Law of continuous change

The first row of the table provides the results for the law of continuous change, which was supported in Phase One of the analysis. Here, in Phase Two the interpretation of the moderated regression model results is that inclusion of the automation tool usage variable enhances support for the law as the new variable is statistically significant at usual levels. Figure 2 graphically depicts the software evolution profile for the portfolio at three levels of automated tool usage: none (shown as a broken line of open boxes), average for the research site (shown as a solid line), and high for the research site (shown as a dotted line of diamonds), all based on starting at the point in the portfolio history when software automation tools were introduced\(^{11}\).

As can be seen in Figure 2 and in Table 7 there are significantly higher activity levels for the portion of the application portfolio with automation. For example, at an average level of

\(^{11}\) All of the figures in this section will follow the same convention, i.e., show the results for the portfolio at the three levels of usage starting from the time when the tool usage was introduced.
automation tool usage the average activity level is approximately 1.5 times that with no use of automation, and at the highest level of use of automation the average activity level is about twice that of no use of automation. In addition, the interaction of automation tool use with AGE (the $\beta_{L4}$ column) is positive and significant, indicating that the activity rate increases at a faster rate with tool usage.

![Figure 2: Activity Levels With / Without Automation (Law of Continuous Change)](image)

For example, at the average usage level of automation tools, activity rates increase at a rate approximately three times that with no automation tool usage. At the highest usage of automation, activity rates increase at almost six times that with no automation. Finally, the interaction of automation tool use with AGE$^2$ (the $\beta_{L5}$ column) is negative and significant, indicating that the increase in activities with tool use is non-linear, i.e., activities increase with tool use, but at a declining rate over time. Finally, the use of automation and its interactions with the AGE variables explain an additional 30% of the variance in activity levels over and above the variance explained
by AGE alone, i.e., as was shown in the Phase One analysis where software automation tool usage was not considered. Therefore, the results support the notion that automation tool usage is influential in increasing the activity level for the software portfolio.

2. Law of continuous growth
The next law, the law of continuous growth in functionality, was supported in Phase One, and, similar to the prior analysis, modeling the effect of automation enhances support for this law as well. Figure 3 graphically depicts the software evolution profile for the portfolio at the three levels of automated tool usage.

Although Figure 3 offers a less dramatic effect than that visualized in Figure 2, Table 7 does document a statistically significant effect of an increased growth in functionality with the use of software automation tools. With no automation the application portfolio grows at a rate of about 15 new modules per month, and the increase is at an increasing rate. With average use of automation the portfolio increases at a rate of 20 modules per month, almost linearly, at a slightly increasing rate. With the highest use of automation the portfolio increases at a rate of 28 modules per month, although the increase itself is at a declining rate. Similar to the law of continuous change, the use of automation and its interactions explain significant variation (55% of the variance) in the growth of portfolio functionality over and above the variance explained by AGE alone. So, our results again demonstrate that automation is influential in its impact on the growth in functionality in terms of the number of modules.

However, it should be noted that, after a sufficiently long time period (at our research site, about ten years), the growth in functionality without automation actually exceeds the growth with automation. With no use of automation the model would suggest that the portfolio would be expected to grow to more than 3,700 modules; with average use of automation, the growth would reach 3,600 modules,
and with highest automation usage the growth would only reach 3,400 modules. One interpretation of this pattern is that over time the use of automation tools is helping portfolio functionality to grow, but at a more “controlled” or “managed” pace, just as suggested by Lehman [28].

![Figure 3: Growth in Functionality With /Without Automation (Law of Continuous Growth)](image)

3. **Law of increasing complexity**

The third law to be tested, the law of increasing complexity, was not supported in Phase One using AGE as the only explanatory variable. However, adding the information about automation tool usage sheds more insight into this initial result. As can be seen in Figure 4, with no automation, the complexity of the portfolio is actually rising over time, increasing by 70%, and almost doubling in ten years.
Figure 4: Software Complexity With / Without Automation (Law of Increasing Complexity)

With average use of automation the complexity level of the portfolio shifts up initially about 20% higher relative to no automation. However, complexity doesn’t increase as rapidly over time, showing only an increase of about 25%, resulting in a total complexity level lower than the complexity level with no automation after ten years. With the highest use of automation the complexity level of the portfolio shifts up initially about 60% higher relative to no automation, but declines substantially over time, ending up after ten years below both the complexity levels with no automation and with average use of automation. The use of automation tools explains about 15% of the variation in complexity (note that only the automation tool variable is significant, not the interactions with AGE or AGE²). So, while the use of automation is associated with a shift up in the complexity level initially, automation does not substantially increase complexity over time. In fact, over the long run, complexity actually declines with high use of automation, completely offsetting the increase in complexity without automation. It is this finding that provides additional insight into the Phase One results and suggests why, when AGE was the only explanatory variable, the law of increasing complexity was not supported. It is the more sophisticated model which
includes the longitudinal empirical data about CASE automation usage that sheds additional light on the long term complexity change.

4. Law of declining quality
The law of declining quality was supported in the Phase One models of this research, although the coefficient on AGE was essentially zero, so that the increase in the number of corrections per module was so small it was almost zero (the number of corrections per module increases at less than .00001 per month per module). As can be seen in Figure 5, with average use of automation tools the intercept shifts up significantly - there are about ten times the number of corrections per module relative to no use of automation. However, there is also a ‘slope effect’ with automation such that the number of corrections per module declines with AGE at about .0002 fewer corrections per month. With the highest use of automation the intercept shifts up 30 times higher relative to no automation (in terms of the number of corrections per module), but again there is a negative coefficient, i.e., slope effect, with highest use of automation as the correction rate falls substantially over time. After ten years the correction rates with automation have declined so much they approach those without automation (correction rate with no automation is .0011; correction rate with average automation is .0038, and the correction rate with high automation is .0088. Thus, over a decade, the correction rate with average use of automation declines by a factor of three, and the correction rate with highest use of automation declines by a factor of four, both relative to correction rates without automation (which remain largely unchanged over the ten year period). In addition, the use of automation tools explains an additional 30% of the variation in the number of corrections per module over and above AGE. Overall the interpretation is that with automation there is initially a shift up in the intercept (a higher level of corrections), but that, over time, the rate of corrections drops substantially. Automation may be helping the organization to keep correction
rates lower than they would otherwise be, given the increased growth in activities and number of modules in the portfolio.

Figure 5: Correction Rates With / Without Automation (Law of Declining Quality)

5. Law of conservation of familiarity
The law of conservation of familiarity was supported in Phase One as the percentage growth rate in the number of modules in the portfolio declined slightly. Again, however, the addition of the automation tool variable provides additional insight. As can be seen in Figure 6, without automation the percentage growth rate actually increases quite sharply - the percentage growth rate increases by a factor of five over ten years (from 2% to 10%). With average use of automation the initial percentage growth rate is higher (4%), but it doesn’t change as much, increasingly slightly to just under 6% over ten years. With higher use of automation the initial percentage growth rate is almost four times that without automation (just over 8%). However, over time, the percentage growth rate with the highest use of automation actually declines quite sharply, going below the percentage growth with no automation and with average automation after approximately three
years. Therefore, the finding in Phase One that the percentage growth rate was declining was correct overall, but did not offer any insight into how this was happening.

Overall, what can be seen from this Phase Two analysis is that the portfolio grows more rapidly (in terms of percentage growth rate) without automation than with automation. The use of automation tools appears to have had a stabilizing effect on the portfolio as a whole, helping to conserve familiarity in the long term.

![Graph showing % Growth Rate With / Without Automation](image)

**Figure 6: % Growth Rate With / Without Automation (Law of Conservation of Familiarity)**

**6. Law of conservation of organizational stability**

The law of conservation of organizational stability was not supported in Phase One. As can be seen in Figure 7 the usage of automation tools significantly increases productivity levels, impacting the work rate as developers become more productive over time. With automation there is a shift up in the level of productivity, just as the literature would suggest, and just as many studies that relied solely on perceptual measures suggested. Thus, there does not appear to be an ‘invariant work rate’ as suggested by the law. Instead, developers are empirically shown to be more productive with automation tools based on this quantitative analysis. With average use of automation the activity
level per developer is almost twice that without automation. With the highest automation tool usage the activity level per developer is almost four times that without automation, i.e., the impact of automation is shown by shifts in the intercept.

![Graph showing work rate with and without automation](image)

**Figure 7: Work Rate With / Without Automation (Law of Conservation of Organizational Stability)**

The work rate, i.e., productivity, increases slightly over time both with and without automation, as indicated by the nearly parallel lines. However, after ten years, the differences between productivity levels with and without automation are about half as much as initially. Overall, what Figure 7 suggests is that the use of automation raises the productivity level without substantially increasing the rate of productivity change over time. The usage of automation tools explains an additional 25% of the variation in work rate besides AGE, so it has a significant impact, but only as an intercept, not substantially increasing the work rate change (slope) over time. This is precisely what we would expect for support of the law on conservation of organizational stability. Referring to early investigations of this law we find that organizational stability refers to an invariant work rate, i.e., a steady level of productive work from a constant level of input resources. Barring any exogenous changes to developers’ work, an invariant work rate implies that their productivity level
will remain unchanged over time. What we have found is that using automation tools has the greatest impact on each developer’s average level of productivity (in terms of the volume of activities performed). Thus, automation tool usage is shown here to raise the level of productivity of the organization to a new constant, *i.e.*, stable, level, but did not substantively change the improvement pace of the work rate over time.

E. **Summary of Phase Two Results**

From the Phase Two analysis where the specific effects of software automation tools are explicitly modeled using moderated regression we can see two main kinds of results. The first kind reflects a greater ability to evaluate and interpret the validation of the laws of software evolution. In the Phase One analysis the laws are tested by using AGE as the explanatory variable. While this is consistent with how the laws have been described, it does not provide significant insight into *why* systems behave in these manners. In addition, in Phase One some laws were shown as not being supported by the data, *e.g.*, the law of increasing complexity. However, the Phase Two analysis, which includes the effect of CASE automation, shows that the use of these tools over time helped the organization to manage the growth of complexity in the software portfolio. Similarly, the invariant work rate hypothesis, which conceivably might be true in an environment without significant technical change, is not supported in this organization because of the adoption and use of automation tools.

The second set of results reveals the direct and long-term impact of automation on variables of interest to both software developers and managers. The use of automation increases the growth in portfolio functionality and work activities accomplished, but at a sustainable rate. Furthermore, automation seems to have a stabilizing effect on growth. For example, although the use of automation increases the level of complexity initially, over time complexity declines sharply with
intensive use of automation. Despite the software system changes expected from software evolution, the intense use of automation significantly improves software quality, i.e., reduces the number of corrections per module, over time.

V. Conclusions
Our analysis of these longitudinal empirical software data shows that automation is helping the software development organization to do more work, to do it more productively, and to increase the functionality of the portfolio, all while managing the rate of growth so that it remains sustainable. All of this occurs simultaneously with complexity that increases at a decreasing rate, despite the increase in maintenance activities and the growing portfolio functionality. Overall, the Phase Two moderated regression model provides a strong validation of the types of effects that many researchers and practitioners have hypothesized (and even hoped) for automation tools, but without longitudinal empirical data it has been nearly impossible to provide quantitative support. In order to demonstrate these effects an organization implementing software automation tools must collect data in a software repository, and do so for a sufficient length of time for the effect to be measured and analyzed. In addition, the use of a moderated regression approach helps to quantify and estimate the precise impacts of the tools on software evolution.

Making major changes to work practices is not easy. All organizations tend to resist changes to the basic core activities they use to survive. Although software development organizations are often perceived to be good at creating change for their customers and users, the irony is that software development organizations may be no better at accepting change than any other organization. However, software development organizations are in a unique position to collect and store great quantities of data recording the effects of changes to standard operating procedures and practices, such as the implementation of automation tools.
It is difficult for software managers to economically justify changes to the processes used in creating and testing software development and maintenance. The effects of decisions about new tools, procedures, languages, operating systems and/or hardware may not be measurable until much later in the life cycle of the software. In addition, most significant changes to software and hardware are relatively expensive. In the past there has been very little evaluation of the long-term outcomes from these costly investments in time and money.

With this research project we have demonstrated the power of analyzing a large empirical dataset of longitudinal data on software systems. This work began by demonstrating support for Lehman’s laws of software evolution, a software engineering benchmark. Then, the archival data have afforded us a natural field experiment to analyze the difference in the behavior of a software portfolio occurring both before and after the implementation of tools for automated code generation. Moreover, our analysis shows how longitudinal empirical software data can be used to test, evaluate and ultimately justify investments in process improvement. Our moderated regression analysis also has allowed us to quantify the particular effects of software process improvement innovations (such as automation) on important software development outcomes – for example, we were able to determine the average productivity levels for developers and the correction rates per module for the organization with and without automation.

Future research that has the goal of understanding the impacts of software process innovations could adopt our approach. First, it is essential for the software development organization to collect data on its systems over a period of time. It can be expected that most innovations require some time to have an effect. As part of the software development effort, repositories such as configuration management systems and project management systems regularly collect data on developer and system characteristics (such as the size, time, and date of change, the developer making the change, and the effort required to make the change) that are archived, but often not
analyzed. The software code in such repositories can also be analyzed, as we have done here, using code analysis tools to, for examples, measure levels of software complexity. Second, the data on system characteristics should be supplemented with measures of the adoption or usage of the particular innovation to be studied. For example, in our study we were able to obtain detailed measures of the extent to which the automation tool had been used in the software portfolio and when the tool was used. This allowed us to conduct a “before and after” analysis of the tool’s impact on the portfolio. Finally, it is helpful to use an analytical technique, such as moderated regression analysis, that can precisely identify the impacts of the process innovation on software development outcomes. Future research studies that adopt this approach can provide important insights that can help software engineers and managers make better development decisions.

VI. REFERENCES


