JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2007; **19**:1–31
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr.342

Research

How software process automation affects software evolution: a longitudinal empirical analysis

Evelyn J. Barry[1], Chris F. Kemerer[2,*,†] and
Sandra A. Slaughter[3]

[1]*Texas A&M University, College Station, TX 77843-3112, U.S.A.*
[2]*University of Pittsburgh, 278A Mervis Hall, Pittsburgh, PA 15260, U.S.A.*
[3]*Carnegie Mellon University, Pittsburgh, PA 15213, U.S.A.*

SUMMARY

This research analyzes longitudinal empirical data on commercial software applications to test and better understand how software evolves over time, and to measure the likely long-term effects of a software process automation tool 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 a series of hypotheses developed by Belady and Lehman about software evolution. We find empirical support for many of these hypotheses, but not all. We then further analyze the data using moderated regression analysis to discern how software process automation efforts at the research site influenced the software evolution lifecycles of the applications. Our results support the claim that automation has enabled the organization to accomplish more work activities with greater productivity, thereby significantly increasing the functionality of the applications portfolio. Despite the growth in software functionality, the analysis suggests that 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 used to reveal the often elusive long-term benefits of investments in software process improvement, and to help managers make more informed resource-allocation decisions. Copyright © 2007 John Wiley & Sons, Ltd.

Received 2 June 2006; Revised 30 November 2006; Accepted 30 November 2006

KEY WORDS: software maintenance; software complexity; software quality; software productivity; computer-aided software engineering (CASE); longitudinal analysis; Lehman's laws of software evolution

*Correspondence to: Chris F. Kemerer, University of Pittsburgh, 278A Mervis Hall, Pittsburgh, PA 15260, U.S.A.
†E-mail: ckemerer@katz.pitt.edu

Contract/grant sponsor: National Science Foundation; contract/grant numbers: CCR-9988227 and CCR-9988315
Contract/grant sponsor: Center for Computational Analysis of Social and Organizational Systems (Research Proposal Award)
Contract/grant sponsor: NSG (IGERT at Carnegie Mellon University)
Contract/grant sponsor: Sloan Software Industry Center
Contract/grant sponsor: Institute for Industrial Competitiveness

Copyright © 2007 John Wiley & Sons, Ltd.

WILEY InterScience® DISCOVER SOMETHING GREAT

1. 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 is no doubt a myriad of reasons for the continuing challenges in software development, one central problem is that it is difficult to determine the cause and effect relationships due to the implementation of different development practices. In part, this is because the consequences from changes and innovations in software development practices are seldom immediate, but instead evolve over time. In addition, an experimental approach, where most of the variables are under the researcher's control, is less feasible when the effects emerge over a long period of time. As a result, it can be difficult for managers to assess the value and therefore motivate the implementation of 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 typically 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. This research also shows 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 the analysis of detailed data from source-code control systems, defect-tracking systems, and archived project documentation. These data represent, in some cases, more than 20 years of software evolution at our datasite. 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 this conjecture due to the difficulty in accessing longitudinal data. In particular, the software evolution laws originally hypothesized by Lehman and published by Belady and Lehman can be evaluated using these data [1] (see also [2]). In a recent paper, Cook *et al*. note that use of the term law is used in the same sense that social scientists use the term to describe general principles that are believed to apply to some class of social situation 'other things being equal', rather than the laws found in sciences such as physics [3]. As a consequence, in this paper we will generally treat the laws as hypotheses, but as these hypotheses are perhaps the earliest and most discussed of the literature in the software evolution area, they are an appropriate starting point for this research.

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. The second research question is *what are the effects of the use of automation in software development tasks*? 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

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

economic perspectives [4]. The 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 a greater use of standardized components, thus increasing software reliability and decreasing the future maintenance costs of software. Finally, automation may reduce the number of less-interesting, more-mechanical tasks software developers are required to perform, thus freeing them to focus on tasks that require more creativity [4,5]. 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 [6].

For all of these reasons software process automation has been widely discussed, debated, critiqued, or promoted. Yet, given that many of the proposed benefits of such automation tend to occur downstream over the lifecycle 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 the empirical evidence of the benefits of automation. However, this is exactly the kind of problem for which longitudinal data could provide an 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 20 years of software repository data from a commercial organization. Our analysis of this data begins with a test of Lehman *et al.*'s laws of software evolution to establish a benchmark. Our 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 software process 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 a greater number of 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. In Section 2 we describe some of the relevant previous research in this area with particular attention paid to the software evolution laws proposed by Lehman *et al.* In Section 3 we describe the first phase of the research where we develop models to test the laws of software evolution. Moderated regression models are then developed in Section 4 to analyze the impact of automated software process automation. 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.

2. PREVIOUS RESEARCH

How are systems expected to behave over time? Although a tremendous amount of anecdotal evidence exists, there is relatively little carefully documented analysis. This is likely due, at least in part, to the difficulty in collecting longitudinal data of this kind. 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 [7]. 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

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

processes in software systems. It will remain difficult to control lifecycle costs of software systems until software evolution is better understood.

2.1. Laws of software evolution

The most well-documented early attempt to study software evolution in a systematic way was conducted by Belady and Lehman beginning in the late 1960s [1]. Their early collaboration continued to expand over the next decade (see [1,7] and [8, pp. 501–522]), and resulted in a set of laws of software evolution [1]. In that seminal paper, Belady and Lehman outlined 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. 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 (also known as invariant work rate) and the law of conservation of familiarity [9]. These two additions describe limitations on software system growth.

Lehman and Belady'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 [8]. 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 [2]. Lehman presents the feedback system law in two assertions. Assertion 1 states that 'The software evolution process for E-type systems, which includes both software development and its maintenance, constitutes a complex feedback learning system'. Assertion 2 states that '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' [10]. Note that 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.

In Table I we summarize this work using the most current names and definitions, and order them into 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 evolution. 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, and therefore we do not reference them by number and have adopted the convention in this paper of referring to the laws by name only. In the review of previous research (Section 2.2) we use the then contemporaneous name in order to be consistent with the prior literature. However, after this review of prior literature we will use only the most modern name for each law as shown in Table I to avoid potential confusion.

2.2. Previous empirical validation studies

In addition to some simulation studies [11,12], a variety of authors have attempted field-based empirical tests involving the laws. In their original studies of software evolution, Belady and Lehman

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

Table I. Software evolution laws [2].

Software evolution laws	Description
Evolution of software system characteristics	
Continuous change	Systems must continually adapt to the environment to maintain satisfactory performance
Continuing growth	Functional content of systems must be continually increased to maintain user satisfaction
Increasing complexity	As systems evolve they become more complex unless work is specifically done to prevent this breakdown in structure
Declining quality	System quality declines unless it is actively maintained and adapted to environmental changes
Organizational/economic resource constraints	
Conservation of familiarity	Incremental rate of growth in system size is constant to conserve the organization's familiarity with the software
Conservation of organizational stability	The organization's average effective global activity rate is invariant throughout the system's lifetime
Meta-laws	
Self regulation	The software evolution processes are self-regulating and promote globally smooth growth of an organization's software
System feedback	Software evolutionary processes must be recognized as multi-level, multi-loop, multi-agent feedback systems in order to achieve system improvement

analyzed observations of 21 releases of 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 module count as their size measure on other software systems to evaluate the same group of laws, and have employed least-squares linear regression and inverse squares in the analysis [1,2,9].

To test the conservation of organizational stability and familiarity, Lehman ran regressions using the change in number of modules as the dependent variable [13]. Results confirmed that the organization performing software maintenance displayed an invariant work rate and conservation of familiarity [9].

In later studies, Chong Hok Yuen, a student of Lehman's, conducted a study on 19 months of on-going 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' [14]. 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 [15,16].

Cooke and Roesch [17] 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.

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

Lehman *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 their eight laws of software evolution: continuous change, increasing complexity, continual growth, conservation of organizational stability, and feedback system [2].

That same year, Gall *et al*. published a conference paper that presented data plots from multiple releases of a telecommunications switching system [18]. They argued that their plots show support for 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 [19]. They also found 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*. used a reverse-engineering approach to track cumulative changes in call and data dependencies across versions of software. They argued that their data support the law of feedback systems [20].

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 tracked growth in lines of code (LOC) from Linux and note that it is 'super linear' (presumably an increasing nonlinear curve), and does not remain constant over time [21]. Aoki *et al*. use data from 360 versions of the open-source system JUN and find that the growth of this system is also at a 'super linear' rate, and because the growth is not constant, the law of conservation of familiarity is not supported [22]. In each of these studies the time series equates a release sequence number with one unit of time. Note, however, that in this paper the results were presented as a concave curve when size is potted 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 used in the study, 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 to researchers.

Finally, a 2004 paper by Paulson *et al*. [23] combines the method used by Gall *et al*. and the open-source orientation of Godfrey *et al*. and Aoki *et al*. Changes in software releases were plotted on a time interval of a 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 [23].

2.3. Summary of previous research

From this summary of previous research a few things seem clear. The first is that Lehman and his colleagues' work on software evolution has merited attention from a variety of researchers. Understanding the behavior of software systems over time is generally seen as a worthy research goal, and this research, which had its origins in the late 1970s, continues to be the extant model on the subject today. However, from an empirical point of view, support for the laws has been mixed as researchers have been generally unable to test more than a small number of the laws, and even then, data limitations tend to severely constrain the analysis.

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

Support has been strongest for the first three laws in Table I, with independent support from Cooke and Roesch for all three and from Gall and Jazayeri for two of the three. Chong Hok Yuen [16] found support for continuous change, continuing growth, and increasing complexity, but not declining quality. Support has been more difficult to find for conservation of familiarity (not supported by Aoki *et al*. [22] or Chong Hok Yuen [15]) and conservation of organizational stability (not supported by Cooke and Roesch [17]), and the meta-laws (self regulation and system feedback) have generally not been subject to the same level of empirical testing as the earlier laws.

Given this previous research, starting the analysis of our empirical data with an evaluation of the laws of software evolution provides a clear benchmark against the previous and current literature. Therefore, the first phase of the analysis will be to treat the laws of software evolution as hypotheses to be tested, in what may be regarded as their most comprehensive independent test to date. This first phase of our research will then form a baseline for the second phase, which uses the longitudinal data to assess the impact of a software process automation tool on how software evolves over time.

3. RESEARCH MODEL: PHASE ONE

3.1. 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 merchandizing.

Most importantly for the research, for more than 20 years the IT group has 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 4000 software modules. This is a rich dataset that affords a rare opportunity to observe software evolution 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 20-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 software process automation on the managerially relevant dimensions of 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 [24]. 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 [24]. Module-level data were aggregated each month to compute

Copyright © 2007 John Wiley & Sons, Ltd.

Table II. Variable measurement.

Variable	Measurement
Age	Age of the portfolio in months
Number of activities	Count of corrections, adaptations, enhancements and new module creations to the portfolio
Module count	Count of the number of modules in the portfolio
Cyclomatics per module	Total cyclomatic complexity of the modules in the portfolio divided by the number of modules
Operands per module	Total operands in the portfolio divided by the number of modules
Calls per module	Total calls in the portfolio divided by the number of modules
Number of corrections per module	Total corrections divided by the number of modules
Percentage growth in module count	Change in number of modules this month divided by the total number of modules last time period
Number of activities per developer	Count of corrections, adaptations, enhancements and new module creations divided by the count of developers making modifications during the month
Number of developers	The number of developers working on the portfolio that month

portfolio-level metrics for size and complexity. One advantage of this dataset is the use of actual modification-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. Similar to other empirical work in this area, we use the number of modules as a measure of size [1,2,9,25].

Table II provides the descriptions of the variables used in our analysis. Descriptive statistics are provided in Table III.

Our data panel consists of 244 monthly observations. To analyze longitudinal data of this sort we use a time-series regression to test each hypothesis. As is common with many datasets for time-series analyses, we found serial correlation [26]. Although the presence of serial correlation means that the estimates will still be unbiased, it does influence the standard errors. If this is not corrected, then the interpretation of the statistical significance could be incorrect. Therefore, we used the widely accepted Prais–Winsten estimators to correct for serial correlation, using the first-order autoregressive (AR1) correction [26]. The results reported in the following analyses have all used the Prais–Winsten estimators as implemented in the software package *Stata*, version 8.

3.2. Phase one modeling

The conception of software evolution is that software systems *change over time*. Therefore, consistent with prior research, the base case version of our model uses a single variable, *system age*, to test the laws [27].

The general form of the model is

$$Y_{Lt} = \alpha_L + \beta_L * AGE_t + \varepsilon_{Lt} \tag{1}$$

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

Table III. Descriptive statistics for variables used in time-series analyses.

Variable	*n*	Mean	Standard deviation	Minimum	Maximum
Number of activities	244	92.54	119.80	0.00	549.00
Number of CASE activities	244	31.40	48.56	0.00	362.00
Number of non-CASE activities	244	61.14	81.34	0.00	367.00
Number of developers	244	15.70	16.06	0.00	53.00
Activities per developer	198	3.72	3.32	0.00	16.64
CASE activities per developer	198	1.21	1.57	0.00	10.97
Non-CASE activities per developer	198	2.51	2.13	0.00	9.92
Module count	244	1029.17	1192.38	5.00	3609.00
Total LOC	244	2 428 035.00	2 880 577.00	10 179.00	8 547 848.00
Total cyclomatics	244	117 116.60	135 267.80	809.00	412 114.80
Total operands	244	1 632 922.00	1 917 948.00	9212.00	5 771 101.00
Total calls	244	27 053.31	32 627.20	112.00	93 033.18
CASE module count	244	3.92	7.18	0.00	44.00
CASE total lines of code	244	21 982.98	42 876.26	0.00	291 669.00
CASE total cyclomatics	244	907.55	1831.29	0.00	13 892.95
CASE total operands	244	13 205.99	25 775.15	0.00	177 003.80
CASE total calls	244	309.79	620.25	0.00	4198.55
Non-CASE module count	244	7 679 918.00	88 581.00	5.00	2653.00
Non-CASE total LOC	244	932 470.90	1 035 712.00	10 179.00	3 184 001.00
Non-CASE total cyclomatics	244	56 101.06	59 852.77	809.00	190 672.50
Non-CASE total operands	244	744 622.80	817 213.00	9212.00	2 548 840.00
Non-CASE total calls	244	5528.80	6184.91	112.00	17443.69
Cyclomatics per module	244	108.41	18.03	48.34	161.80
Operands per module	244	1383.99	306.24	570.91	1910.54
Calls per module	244	17.35	11.35	4.34	38.25
CASE cyclomatics per module	120	239.17	33.72	205.59	432.00
CASE operands per module	120	3474.81	448.22	2931.97	6024.00
CASE calls per module	120	85.37	11.09	77.80	146.00
Non-CASE cyclomatics per module	244	87.74	20.94	48.33	161.80
Non-CASE operands per module	244	1072.09	198.61	570.91	1842.40
Non-CASE calls per module	244	7.58	3.52	4.38	22.40
Total corrections	244	11.02	14.00	0.00	60.00
Corrections per module	244	0.01	0.01	0.00	0.08
CASE corrections per module	120	0.02	0.02	0.00	0.18
Non-CASE corrections per module	244	5.3×10^{-3}	0.01	0.00	0.03
Percentage growth in modules	243	0.03	0.08	0.00	1.03
Percentage growth in CASE modules	119	0.10	0.54	0.00	5.67
Percentage growth in non-CASE modules	243	0.03	0.08	0.00	1.03
Number of CASE modules	244	261.18	326.56	0.00	956.00
CASE tool use	244	0.13	0.14	0.00	0.37

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

Table IV. Summary of phase one results ($n = 244$, $p < 0.05$ in bold; $n = 198$ for conservation of organizational stability).

Hypothesis	Dependent variable	Adjusted R^2	β_L	Standard error	t	$P > \|t\|$	Supported?
Continuous change	Number of activities	0.2294	1.4312	0.1656	**8.64**	**0.000**	**Yes**
Continuous growth in functionality	Module count	0.0126	14.8314	1.1346	**13.07**	**0.000**	**Yes**
Increasing complexity	Cyclomatics per module	0.1744	−0.0961	0.1072	−0.90	0.371	No
	Operands per module	0.0895	−0.3838	1.8770	−0.20	0.838	No
	Calls per module	0.0100	0.0158	0.0447	0.35	0.724	No
Declining quality	Number of corrections per module	0.0282	0.000 06	0.000 02	**2.91**	**0.004**	**Yes**
Conservation of familiarity	Percentage growth in module count	0.0212	−0.0002	0.0001	**−2.51**	**0.013**	**Yes**
Conservation of organizational stability	Number of activities per developer	0.1902	0.0334	0.0039	**8.65**	**0.000**	**No**

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 [26,28].

In all of the models in this paper we use the customary notation α to represent the intercept term, β to represent the coefficients of the independent variables, and ε to represent the error term. Table IV provides a summary of the estimation results for each hypothesis. (As explained in Section 4.6, all regressions were also run with a robust standard error formulation (not shown here), which generally strengthens the statistical significance of the results.)

3.2.1. Continuous change hypothesis

Continuous change states that '[a] system must be continually adapted else it becomes progressively less satisfactory in use' [29]. For the dependent variable, we use a count of all changes and additions to the software portfolio. Note that the mix of software maintenance activities (12% corrective, 75% enhancement, 11% adaptive) is similar during the lifetime of the software portfolio (a comparison of the average percentage mix of activities year-by-year suggests no significant differences). Thus, although certain types of activity may require more or less effort than others, because the mix of activities remains similar over the lifetime of the software portfolio, there should be no bias in our results due to changes in the types of activities being performed. Our results reveal that the coefficient on the AGE variable is positive and significant, supporting the hypothesis of continuous

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

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.

3.2.2. Continuous growth hypothesis

Continuous growth in functionality 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 continuous change. *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.

3.2.3. Increasing complexity hypothesis

In order to test the increasing complexity hypothesis we use the widely recognized McCabe cyclomatic complexity per module as a complexity metric [30]. However, we also examine two other measures of software complexity: operands per module [31] and calls per module. These metrics were specifically chosen as they represent contemporaneous metrics for the software examined [32]. However, despite using multiple metrics to guard against the possibility that any results would be somehow metric-specific, we did not find empirical support for this hypothesis for *any* of the three different measures of software complexity because 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 original 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.

3.2.4. Declining quality hypothesis

The fourth and final software system characteristic law 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 empirical support for this law, as the coefficient on *AGE* is positive and statistically significantly different from zero. However, in practical terms, the small size of the estimated coefficient (0.000 06), suggests that the corrections per module increase only very slightly with age.

3.2.5. Conservation of familiarity hypothesis

The next set of hypotheses contains those relating to operational, or economic resource, constraints. Conservation of familiarity is tested using the percentage growth in the number of modules each month. Note that this is different from the test 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, *AGE* explains about 2% of the variation in percentage growth of modules over time, but the coefficient on *AGE* is

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

very small (close to zero), implying that the decrease in the percentage growth rate with *AGE* is very small.

3.2.6. Conservation of organizational stability hypothesis

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. As we have noted earlier, there are no significant differences in the average mix (type) of activities completed each year over the lifetime of the software portfolio. A means test suggests that there are also no significant differences in the size, on average, of each activity completed each year in the lifetime of the software portfolio (the average is about one thousand new SLOC per activity). This suggests that activity counts are comparable across time periods (i.e., do not systematically vary by type and size) and can be used to assess the work accomplished 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. (Note that although the estimated coefficient is highlighted as statistically significant, it is statistically significant in the *opposite* direction predicted by the law.) *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.

3.2.7. The 'meta-laws'

Finally, two of the laws of software evolution can actually be seen as 'meta-laws' (self regulation and system feedback). The law of self regulation states that global E-type system evolution processes are self-regulating [29]. The other 'meta-law' is known as the feedback system, which states that evolution processes are multi-level, multi-loop, multi-agent feedback systems [29]. 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 controllable pace, without significantly increasing or decreasing the complexity and quality of the portfolio.

3.3. Extension of phase one analysis: nonlinear model

Recent research has hypothesized that some systems display a 'super-linear' growth [21–23]. The empirical results appear to be mixed. Aoki *et al*. found super-linear growth in the releases of JUN [22], while others have found that software grows at a linear rate when the source-code growth rate is measured over elapsed time rather than the growth between release sequence numbers [23].

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

Given these mixed results, we extended our model to check the quadratic form to see how our estimated regressions 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 nonlinear effect of *AGE* on the dependent variable of the form

$$Y_{Lt} = \alpha_L + \beta_{L1} * AGE_t + \beta_{L2} * AGE_t^2 + \varepsilon_{Lt} \tag{2}$$

where the variables are the same as in the previous model (1) with the addition of an AGE^2 term which is added to allow for a nonlinear relationship in the software evolution pattern exhibited over time. 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 V. This further analysis does not change any of the main results of the earlier section, as hypotheses that were supported in the simpler, linear formulation continue to be supported with the nonlinear model, and *vice versa*. Adding the quadratic term increases the variation explained in the dependent variable (in terms of adjusted R^2) by significant amounts for the second hypothesis tested (continuous growth), but has relatively little effect on the tests of the others. (Note that the quadratic term for the hypothesis of increasing complexity is nearly significant at the 0.05 level for one of the three complexity measures tested.) This suggests that system change and growth over time may be more accurately modeled as a nonlinear function (in this case, specifically, a quadratic function), as fit improves for some models, and does not appreciably decline for the others.

4. 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. The 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. Of course, as is typical in empirical research studies, our results are presented and properly interpreted as 'all else being equal'. We believe that this is reasonable and appropriate given the significant number of observations in the dataset.

4.1. Prior research on software process automation evaluation

Early in its history, software process automation acquired the label of CASE (computer-aided software engineering) tools. Although there is a large amount of 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

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

Table V. Phase one results: Nonlinear model ($n = 244$, $p < 0.05$ in bold; $n = 198$ for conservation of organizational stability).

Hypothesis	Dependent variable	Adjusted R^2	β_{L1} Standard error t-statistic p-value	β_{L2} Standard error t-statistic p-value
Continuous change	Number of activities	0.2508	**1.4338** **0.1579** **9.08** **0.000**	0.0037 0.0025 1.51 0.133
Continuous growth	Module count	0.2976	169.2190 95.4205 1.77 0.077	**0.2385** **0.0985** **2.42** **0.016**
Increasing complexity	Cyclomatics per module	0.1559	−0.1131 0.1218 −0.93 0.354	0.0026 0.0014 1.88 0.061
	Operands per module	0.1031	−0.3261 1.8121 −0.18 0.857	0.0326 0.0187 1.74 0.083
	Calls per module	0.0243	0.0166 0.0431 0.39 0.700	0.0007 0.0004 1.93 0.054
Declining quality	Number of corrections per module	0.0420	**0.0001** **0.0000** **3.15** **0.002**	-5.51×10^{-7} 3.09×10^{-7} −1.79 0.075
Conservation of familiarity	Percentage growth in module count	0.0190	**−0.0002** **0.0001** **−3.08** **0.002**	9.14×10^{-7} 1.13×10^{-6} 0.81 0.420
Conservation of organizational stability	Number of activities per developer	0.1944	**0.0329** **0.0039** **8.48** **0.000**	0.0001 0.0001 1.26 0.210

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

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 reliability of the software development process and 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 tends to produce [33].

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 with 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 [34–38]. Some early work on quantitative measures was promising for automated software tool usage, e.g., with respect to aiding software reuse [39]. 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 concern 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.*' [40, p. 36]

How does the innovation of software process automation fit into the view of software evolution? One starting point can be found in the work of Lehman himself, in a 1990 conference paper [5]. In it he 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 [5, p. 240]. 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 number of errors, less rapid growth of system complexity, more reliable operation, less down time.*' [5, p. 243]

Finally, and not surprisingly, because the underlying economic conditions that produced the interest in, and demand for, software process automation tools remain relatively unchanged today, there continues to be current interest in tool usage and its impact on performance (e.g., Limayem *et al.* [37]). This interest can be expected to persist, especially as organizations continue to look for ways to control and reduce software labor costs, for example, the current offshore outsourcing phenomenon. The software repository data at the research site affords us the opportunity to investigate the *measured* impact of one of these tools over time, in contrast to most of the literature described above, which was limited to surveys of participants' *perceptions* of the impact of the tools.

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

4.2. Software process automation at the data site

The IT group at our research site introduced an automation tool approximately half-way through our data collection period (month 125 of 244). Their choice was the implementation of a commercially available COBOL source-code generator, hereafter referred to as the tool. The company had gone through a series of mergers and acquisitions that resulted in a dramatically changed set of application software systems. This tool was 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*. noted that

> '*The greatest benefit of ... tools lies in their capability of generating computer module codes. This capability can greatly improve developers' productivity as well as the quality, reusability, portability, and scalability of applications*.' [37]

One question which may arise is whether the ensuing software evolution patterns are triggered by changes in the corporate environment at our research site or by use of the tool. Fortunately, we are able to distinguish the effects of tool use from general shifts in the environment as we can identify which particular modules in the portfolio were developed using the tool and which were not. As some modules are created using the tool and some are not, both before and after the points in time when the organization is experiencing changes in its environment, their identification serves as a natural control that allows us to discern the effects of variations in the use of the tool to variations in software evolution patterns.

4.3. Data on software process 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 $TOOL_t$ 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 the numbers of modules. When the tool is introduced at month 125 the portfolio is growing very rapidly. Although the percentage of the portfolio created using an automation tool peaks within the first two years of its introduction, more than one-quarter of all modules are developed with tool support.

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

4.4. Phase two modeling and results

To evaluate the impact of automation on the evolution of the software portfolio, we adopt a moderated regression approach [41,42]. 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

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

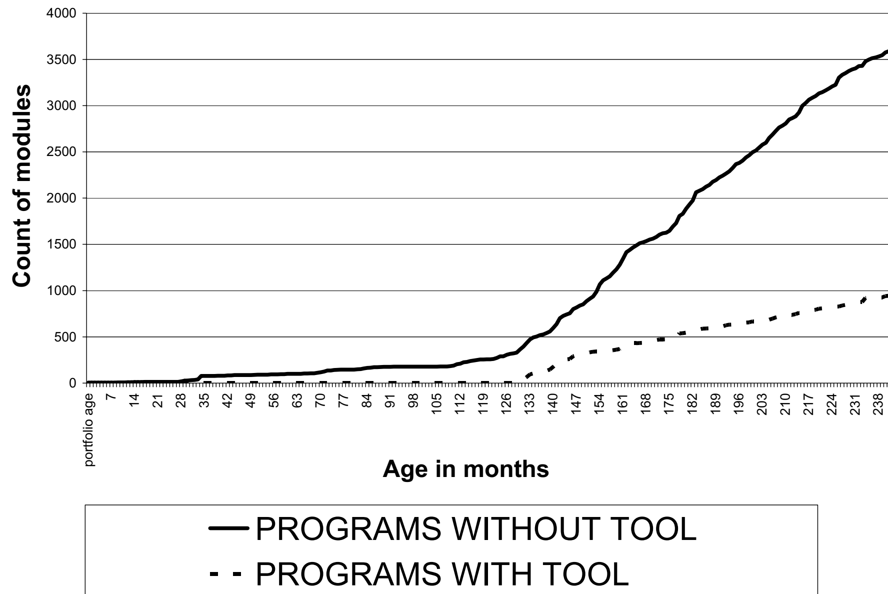

Figure 1. Portfolio growth and the growth of modules developed using the automation tool.

the interaction of two or more independent variables. In moderated regression, the researcher is attempting to understand more than just the linear and additive effects of some predictor variables on some outcome variable, and wants to understand whether the value of an outcome variable depends *jointly* upon the value of the predictor variables [43].

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

$$Y_{Lt} = \alpha_L + \beta_{L1} * AGE_t + \beta_{L2} * AGE_t^2 + \beta_{L3} * TOOL_t + \beta_{L4} * TOOL_t * AGE_t + \beta_{L5} * TOOL_t * AGE_t^2 + \varepsilon_{Lt} \quad (3)$$

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.

Interpreting the results of the model can be done as follows. For example, the use of automation is associated with a higher average number of activities performed per programmer (relative to no automation) if the coefficient on the *TOOL* variable, β_{L3}, is positive and significant. A negative and significant coefficient on this variable indicates that the use of automation is associated with a lower average number of activities performed per programmer (relative to no automation). We can also look at the interaction effects to see if automation either amplifies or counteracts the effects of *AGE*. For example, automation could significantly increase the number of activities performed per programmer (this would be indicated if the coefficient on the interaction of $AGE \times TOOL$ were positive and significant). A negative coefficient on this interaction would indicate that the use of automation

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

Table VI. Descriptive statistics for the software module characteristics as a function of automation.

	Developed with automation tool ($n = 956$ modules)		Developed manually ($n = 2653$ modules)	
Module dimension	Mean	Standard deviation	Mean	Standard deviation
Lines of code	5840.3	497.4	1292.9	183.6
Total cyclomatics	239.2	33.7	87.7	20.9
Total operands	3474.8	488.2	1072.1	198.6
Total calls	85.4	11.1	7.6	3.5

Table VII. Descriptive statistics for the tool automation variables.

Variable	Observations	Mean	Standard deviation	Minimum	Maximum
Portfolio *AGE* (in months)	244	122.5	70.58	1	244
CASE tool use	244	0.13	0.14	0	0.37

significantly decreases the number of activities performed per programmer. Alternatively, automation could be seen as causing the portfolio to grow at an increasing or decreasing (nonlinear) rate (this could be observed if the coefficient on the $AGE^2 \times TOOL$ interaction were significant). For example, a positive (negative) coefficient on this interaction indicates that use of automation changes the number of activities performed per programmer at an increasing (decreasing) rate. Table VII shows the descriptive statistics for the *AGE* and *TOOL* automation variables.

Table VIII presents the results of testing the hypotheses on software evolution taking into account the specific effect of a software process automation tool.

4.4.1. Continuous change hypothesis

The first row of Table VIII provides the results for the continuous change hypothesis, 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 interaction variables enhances support for the hypothesis as these new variables are statistically significant at the usual levels. Figure 2 graphically depicts the software evolution profile for the portfolio at three levels of automated tool usage: none (shown as a curve of open boxes), average for the research site (shown as a full curve), and high for the research site (shown as a curve of full diamonds), all based on starting at the point in the portfolio history when the software process automation tool was introduced. (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.) We used the mean level of tool use at the research site (12.8%) for the average, no tool use (0%) for none, and the maximum level of tool use at the research site (37%) for high.

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

Table VIII. Results of phase two analysis ($n = 244$, $p < 0.05$ in bold; $n = 198$ for conservation of organizational stability).

Law	Dependent variable	Adjusted R^2	*AGE* (β_{L1}) (standard error) *t*-statistic *p*-value	*AGE*^{2} (β_{L2}) (standard error) *t*-statistic *p*-value	*TOOL* (β_{L3}) (standard error) *t*-statistic *p*-value	*TOOL* ∗ *AGE* (β_{L4}) (standard error) *t*-statistic *p*-value	*TOOL* ∗ *AGE*^{2} (β_{L5}) (standard error) *t*-statistic *p*-value
Continuous change	Number of activities	0.5393	**2.7979** **(0.8051)** **3.48** **0.001**	**−0.0119** **(0.0057)** **−2.08** **0.039**	110.7575 (162.3078) 0.68 0.496	**16.5180** **(5.9278)** **2.79** **0.006**	**−0.1319** **(0.0493)** **−2.68** **0.008**
Continuing growth	Module count	0.5534	**19.9264** **(1.6373)** **12.17** **0.000**	**0.0546** **(0.0096)** **5.71** **0.000**	−285.5146 (187.8702) −1.52 0.130	**32.7173** **(9.5351)** **3.43** **0.001**	**−0.3068** **(0.0865)** **−3.55** **0.000**
Increasing complexity	Cyclomatics per module	0.2945	0.015 703 (0.3252) 0.05 0.962	0.0017 (0.0023) 0.74 0.461	**134.7242** **(51.9728)** **2.59** **0.010**	0.6451 (2.4565) 0.26 0.793	−0.0195 (0.0199) −0.98 0.327
	Operands per module	0.3556	2.0575 (3.6104) 0.57 0.569	0.0174 (0.0255) 0.68 0.496	**2166.014** **(575.5422)** **3.76** **0.000**	10.3776 (27.2254) 0.38 0.703	−0.3094 (0.2209) −1.40 0.163
	Calls per module	0.6730	0.0787 (0.0506) 1.56 0.121	0.0002 (0.0003) 0.57 0.566	**73.485** **(6.4603)** **11.37** **0.000**	0.5842 (0.3237) 1.80 0.072	**−0.0089** **(0.0029)** **−3.10** **0.002**
Declining quality	Number of corrections per module	0.3549	**−0.0002** **(0.0001)** **−1.99** **0.048**	5.98×10^{-7} (5.90×10^{-7}) 1.01 0.312	**0.1047** **(0.0167)** **6.26** **0.000**	**−0.0013** **(0.0006)** **−2.17** **0.031**	5.18×10^{-6} (5.90×10^{-6}) 1.02 0.310
Conservation of familiarity	Percentage growth in module count	0.0200	−0.0005 (0.0007) −0.67 0.502	5.04×10^{-6} (5.11×10^{-6}) 0.99 0.325	0.1690 (0.1444) 1.17 0.243	−0.0045 (0.0053) −0.86 0.391	5.26×10^{-7} (0.0000) 0.01 0.991
Conservation of organizational stability	Number of activities per developer	0.4448	0.0123 (0.0217) 0.57 0.571	0.0001 (0.0002) 0.61 0.541	**13.703** **(4.2443)** **3.23** **0.001**	−0.0177 (0.1569) −0.11 0.910	−0.0002 (0.0014) −0.15 0.883

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

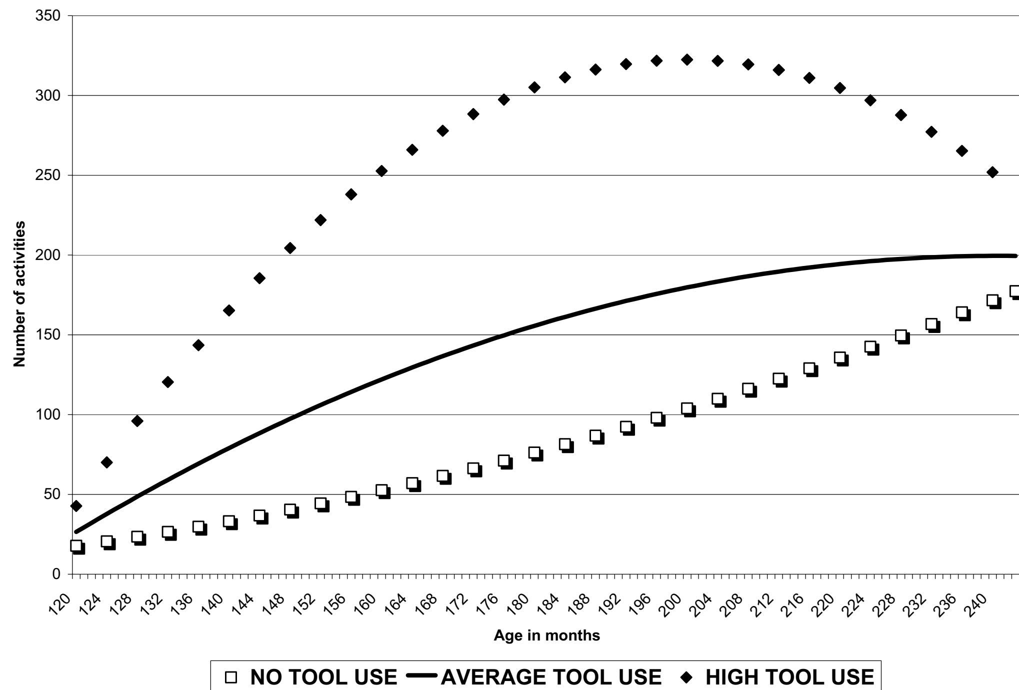

Figure 2. Activity levels with/without automation (law of continuous change).

As can be seen in Figure 2 and in Table VIII there are significantly higher activity levels for the portion of the application portfolio developed using process automation. For example, at an average level of 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 β_{L4} column) is positive and significant, indicating that the activity rate increases at a faster rate with tool usage.

For example, at the average usage level of the automation tool, 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 β_{L5} column) is negative and significant, indicating that the increase in activities with tool use is nonlinear, i.e., activities increase with tool use, but at a declining rate over time. 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 process 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.

Copyright © 2007 John Wiley & Sons, Ltd.

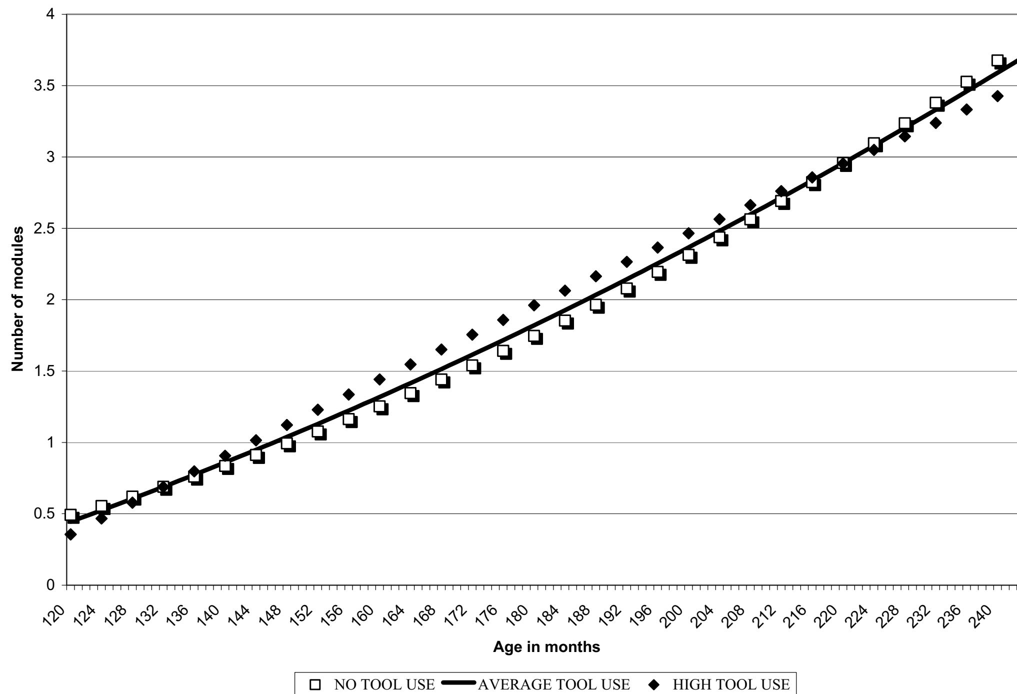

Figure 3. Growth in functionality with/without automation (law of continuous growth).

4.4.2. Continuous growth hypothesis

The hypothesis of continuous growth was supported in phase one and, similar to the previous analysis, modeling the effect of automation enhances support for this hypothesis as well. Figure 3 graphically depicts the software evolution profile for the portfolio at the three levels of automated tool usage.

Although Figure 3 (which shows the relationship between the number of modules (thousands) and time) offers a less dramatic effect than that visualized in Figure 2, Table VIII does document a statistically significant effect of an increased growth in size (functionality) with the use of the software process automation tool. 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 appears to be influential in its impact on the growth in functionality in terms of the number of modules.

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

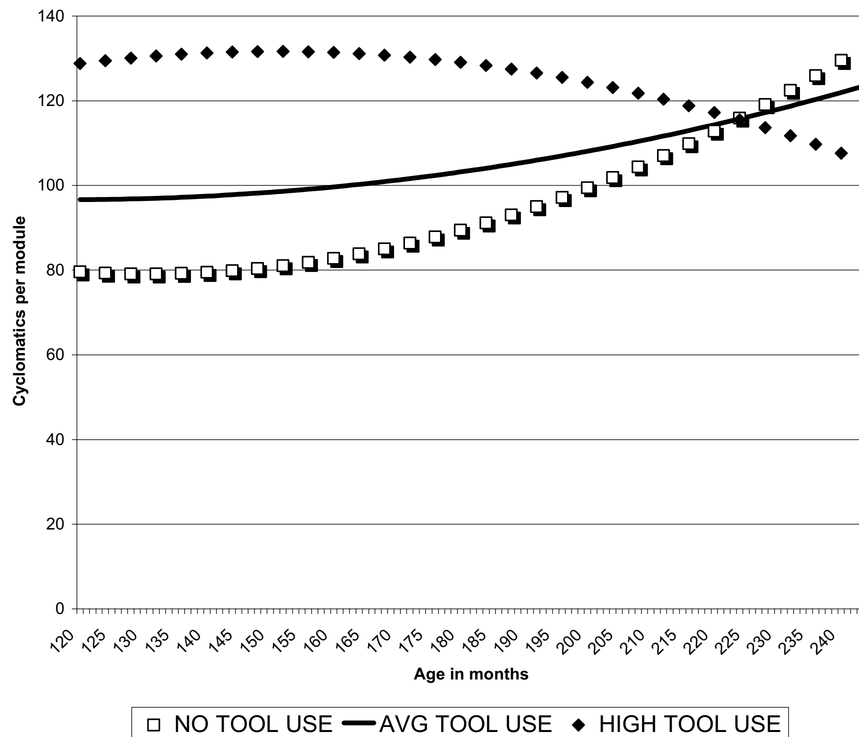

Figure 4. Software complexity with/without automation (law of increasing complexity).

However, it should be noted that, after a sufficiently long time period (at our research site, about 10 years), the growth in functionality without automation actually exceeds the growth with automation. With no use of automation, the model suggests that the portfolio would be expected to grow to more than 3700 modules; with average use of automation, the growth would reach 3600 modules; and with highest automation usage, the growth would only reach 3400 modules. One interpretation of this pattern is that over time the use of an automation tool is helping portfolio functionality to grow, but at a more controlled or managed pace, just as suggested by Lehman [5].

4.4.3. *Increasing complexity hypothesis*

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 10 years.

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

With average use of automation, the complexity level of the portfolio shifts up initially about 20% higher relative to no automation. However, complexity does not 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 10 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 10 years, below both the complexity level with no automation and that with average use of automation. The use of an automation tool explains about 15% of the variation in complexity.

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, in 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 tool automation usage that sheds additional light on the long-term complexity change.

4.4.4. Declining quality hypothesis

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 0.000 01 per month per module). As can be seen in Figure 5, with average use of an automation tool the intercept shifts up significantly: there are about 10 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 0.0002 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 the highest use of automation as the correction rate falls substantially over time. After 10 years the correction rates with automation have declined so much that they approach those without automation (correction rate with no automation is 0.0011, correction rate with average automation is 0.0038, and the correction rate with high automation is 0.0088). Thus, over a decade, the correction rate with an average use of automation declines by a factor of three, and the correction rate with the highest use of automation declines by a factor of four, both relative to correction rates without automation (which remain largely unchanged over the 10-year period). In addition, the use of an automation tool 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.

4.4.5. Conservation of familiarity hypothesis

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

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

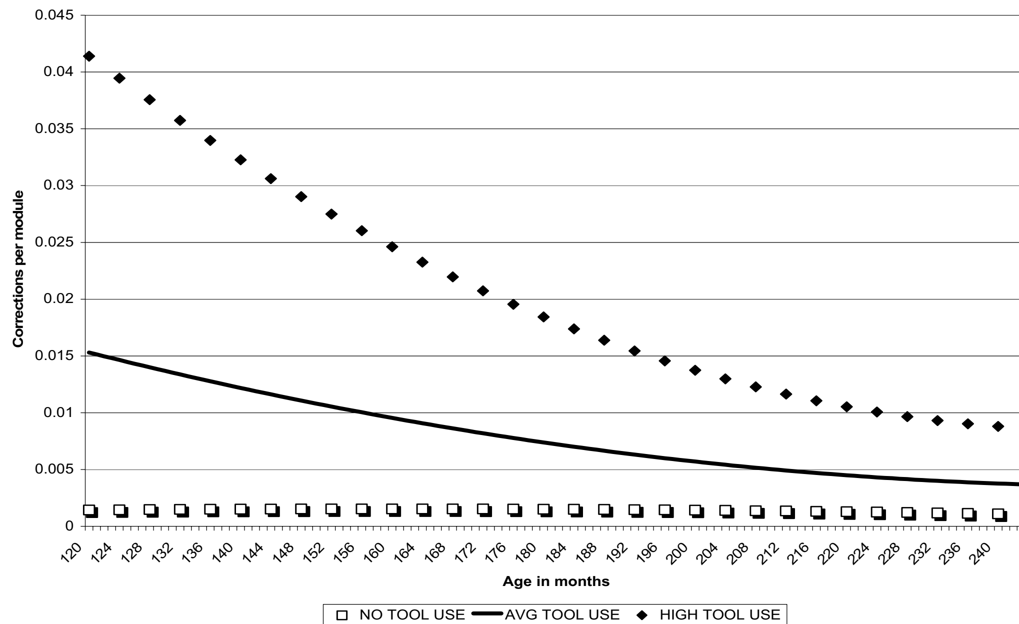

Figure 5. Correction rates with/without automation (law of declining quality).

over 10 years (from 2% to 10%). With average use of automation the initial percentage growth rate is higher (4%), but it does not change as much, increasingly slightly to just under 6% over 10 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 an automation tool appears to have had a stabilizing effect on the portfolio as a whole, helping to, in Lehman's terms, 'conserve familiarity' in the long term.

4.4.6. *Conservation of organizational stability hypothesis*

Conservation of organizational stability was not supported in phase one. As can be seen in Figure 7 the usage of an automation tool 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

Copyright © 2007 John Wiley & Sons, Ltd.

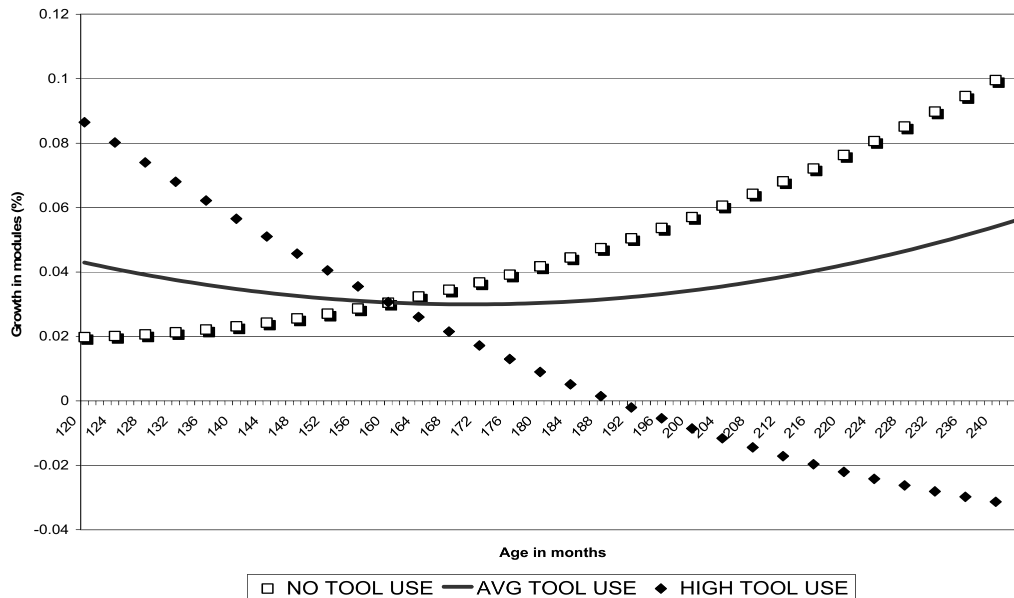

Figure 6. Percentage growth rate with/without automation (law of conservation of familiarity).

perceptual measures suggested. Thus, there does not appear to be an ‘invariant work rate’ as suggested by the conservation of organizational stability hypothesis. Instead, developers are empirically shown to be more productive with an automation tool based on this quantitative analysis. With an 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.

The work rate, i.e., productivity, increases slightly over time both with and without automation, as indicated by the nearly parallel lines. However, after 10 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 an automation tool explains an additional 25% of the variation in work rate besides *AGE*, so it has a significant impact, but only as an intercept (i.e., raising the productivity level), not substantially increasing the slope or rate of change in productivity over time. This is precisely what we would expect in 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 an automation tool has the greatest impact on each developer’s average level of productivity (in terms of the number of activities performed). Thus, automation tool usage is shown here to raise the level of productivity of

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

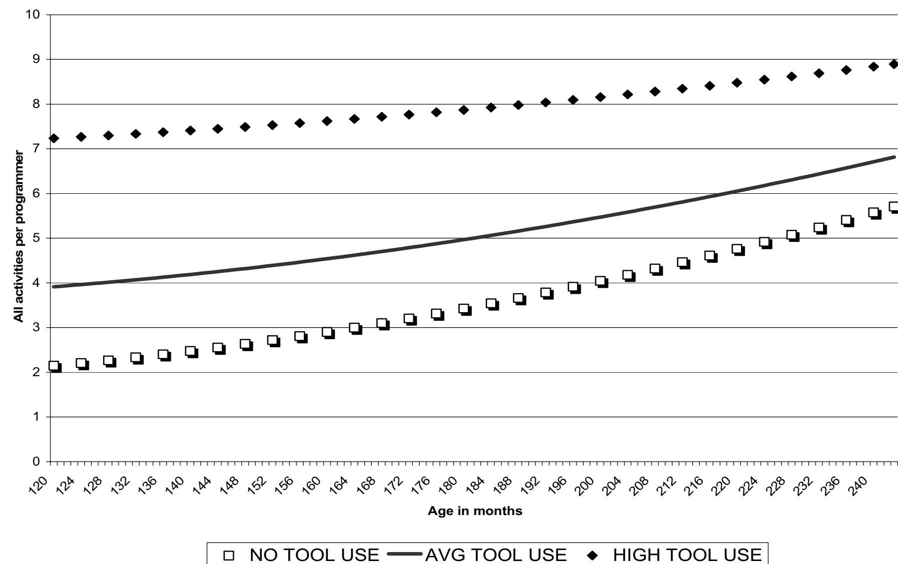

Figure 7. Work rate with/without automation (law of conservation of organizational stability).

the organization to a new constant, i.e., stable, level, but did not substantively change the improvement pace of the work rate over time.

4.5. Summary of phase two results

From the phase two analysis where the specific effects of software process an automation tool 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 test of the hypotheses on software evolution. In the phase one analysis, the hypotheses are tested by using *AGE* as the explanatory variable. While this is consistent with how the hypotheses have been described, it does not provide significant insight into *why* systems behave in these manners. In addition, in phase one some hypotheses 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 software process automation, shows that the use of a tool 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 an automation tool.

The phase two 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 rate that is sustainable for the organization. Furthermore, automation seems to have a stabilizing effect on growth. For example,

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

although the use of automation increases the level of complexity initially, over time complexity declines sharply with intensive use of automation. The intensive use of automation significantly *improves* software quality (i.e., reduces the number of corrections per module) over time despite the software system changes resulting from continued software evolution.

4.6. Sensitivity analysis

Appropriate interpretation of the results of estimates from regression models assumes that the data meet a variety of standard assumptions concerning the distribution of the data. As the nature of the phenomenon of software evolution is one of behavior over time, we collected a time series dataset. Time series data typically suffer from some degree of serial correlation, i.e., values in period $t + 1$ are unlikely to be independent from their value in period t. Therefore, as described above, we estimated all of the models using the Prais–Winsten correction for serial correlation.

There are a variety of other standard assumptions including normality, heteroskedasticity, and the absence of influential outliers. In order to accommodate possible violations of these assumptions we re-ran all of the regressions using the robust variance estimators of Huber and White (also known as the sandwich estimators) [44–46]. There are 24 regression models (for five of the laws there are three models each, linear, quadratic, and interaction, for a sub-total of 15, and for the law on increasing complexity there are three models times three different measures of complexity which is equal to nine models; $15 + 9 = 24$). We find that these robust models yield essentially the same results: in one case the statistical significance is worse, 15 are essentially unchanged, and in eight cases the results are actually stronger in terms of improved statistical significance. (In particular, in the robust model the coefficients for AGE^2, *TOOL*, and $AGE \times TOOL$ are statistically significant at the 0.05 level for the interaction model for the law of conservation of familiarity, which allows for their interpretation as shown in Figure 6.) In the single inferior case (law of declining quality, interaction model) only one of the original three variables is significant at the 0.05 level. However, even in that case the overall model remains statistically significant ($F = 42.79$, $p < 0.001$).

We also test for collinearity using the Belsley–Kuh–Welsch (BKW) condition index [47]. The value for model 1 (*AGE* only) was 1.0, for model 2 (*AGE* and AGE^2) 2.618, and for model 3 (fully interactive) 22.3. All of these values are beneath the BKW threshold value of 30 for potentially undesirable collinearity. (The third model is naturally the highest given the inclusion of interaction terms, e.g., some collinearity would be expected between *AGE* and $TOOL \times AGE$, *TOOL* and $TOOL \times AGE^2$, etc.) However, all values for this index are well below the threshold level. Therefore, the estimated values of the coefficients in all of the models can be interpreted with confidence with regard to the desirable absence of collinearity.

5. 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

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

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 process 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 lifecycle 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 a tool 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 using actual data.

Of course, no such empirical study can be comprehensive, nor can it satisfy every reader's questions. Although we use a large dataset, and one that is unusual in software engineering owing to its significant longitudinal nature, all of our data are from a single organization. It is certainly possible that similar data, should it become available from another research site, could support different results. Of course, use of data from a single organization also offers the benefits of essentially controlling for a number of other factors that might otherwise vary were data to be collected from multiple organizations, e.g., perhaps through a survey instrument. Single organization data are also more likely to be collected in a consistent manner; again, a factor that is less likely in a multi-organization study unless very strict research controls are maintained.

Testing software evolution is inherently a dynamic exercise that requires longitudinal data. However, any longitudinal dataset will include the effects of the passage of time. In this study we are interested in the impact of an automated tool, and are able to identify which modules were developed using the tool and which were not. Therefore, and because it is the goal of such tools to improve performance

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

in the ways in which we have measured, there is a strong belief that the effects that are shown are due to the tool use. However, as in all such empirical studies, it is always possible that there is some other unmeasured variable that coincides exactly or in a significant manner with the tool variable and is therefore also or even primarily responsible for the observed effect. While we do not believe there to be such a phenomenon at this research site, replication of this study within another organization or over a different 20-year time period would certainly be of value in confirming the results we have shown here.

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 a code-analysis tool to, for example, 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 help to 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.

ACKNOWLEDGEMENTS

This research is 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.

REFERENCES

1. Belady L, Lehman M. A model of large program development. *IBM Systems Journal* 1976; **15**(3):225–252.
2. Lehman MM, Ramil JF, Wernick PD, Turski WM. Metrics and laws of software evolution—the nineties view. *Proceedings of the 4th International Software Metrics Symposium (Metrics '97)*. IEEE Computer Society Press: Los Alamitos CA, 1997; 20.
3. Cook S, Harrison R, Lehman M, Wernick PD. Evolution in software systems: Foundations of the SPE classification scheme. *Journal of Systems Management and Evolution* 2006; **18**(1):1–35.
4. Kemerer CF. An agenda for research in the managerial evaluation of computer-aided software engineering tool impacts. *Proceedings of the 22nd Hawaii International Conference on System Sciences*, vol. 2. IEEE Computer Society Press: Los Alamitos CA, 1989; 219–228.
5. Lehman MM. Software uncertainty and the role of CASE in its minimization and control. *Proceedings of the 5th Jerusalem Conference on Information Technology*. IEEE Computer Society Press: Los Alamitos CA, 1990.
6. Brooks FJ. *The Mythical Man-Month* (anniversary edn). Addison-Wesley: Reading MA, 1995; 322 pp.
7. Lehman MM, Ramil JF. Software evolution—background, theory, practice. *Information Processing Letters* 2003; **88**(1–2):33–44.

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

8. Belady LA, Lehman MM. *Program Evolution: Processes of Software Change*. Academic Press: London, 1985; 538 pp.
9. Lehman MM. On understanding laws, evolution and conservation in the large program life cycle. *Journal of Systems and Software* 1980; **1**(1):213–221.
10. Lehman MM. Feedback in the software evolution process. *Information and Software Technology* 1996; **38**(11):681–686.
11. Chatters B, Lehman M, Ramil J, Wernick P. Modelling a software evolution process. *Software Process: Improvement and Practice* 2000; **5**(2–3):91–102.
12. Wernick P, Lehman M. Software process dynamic modelling for FEAST/1. *Journal of Systems and Software* 1999; **46**(5):193–201.
13. Lehman M. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE (Special Issue on Software Engineering)* 1980; **68**(9):1060–1076.
14. Chong Hok Yuen CKS. An empirical approach to the study of errors in large software under maintenance. *Proceedings IEEE International Conference on Software Maintenance*. IEEE Computer Society Press: Los Alamitos CA, 1985; 96–105.
15. Chong Hok Yuen CKS. A statistical rationale for evolution dynamics concepts. *Proceedings IEEE 3rd International Conference on Software Maintenance*. IEEE Computer Society Press: Los Alamitos CA, 1987; 156–164.
16. Chong Hok Yuen CKS. On analyzing maintenance process data at the global and detailed levels: A case study. *Proceedings IEEE 4th International Conference on Software Maintenance*. IEEE Computer Society Press: Los Alamitos CA, 1988; 248–255.
17. Cooke CR, Roesch A. Real-time software metrics. *Journal of Systems and Software* 1994; **24**(3):223–237.
18. Gall H, Jazayeri M. Klosch RR, Trausmuth G. Software evolution observations based on product release history. *Proceedings of the International Conference on Software Maintenance*. IEEE Press: Piscataway NJ, 1997; 160–166.
19. Lehman MM, Perry DE, Ramil JF. On evidence supporting the FEAST hypothesis and the laws of software evolution. *Proceedings of the 5th International Software Metrics Symposium*. IEEE Press: Piscataway NJ, 1998; 84–88.
20. Burd E, Bradley J, Davey J. Studying the process of software change: An analysis of software evolution. *Proceedings of the 7th Working Conference on Reverse Engineering*. IEEE Computer Society Press: Los Alamitos CA, 2000; 232–239.
21. Godfrey MW, Tu Q. Evolution in open source software: A case study. *Proceedings IEEE International Conference on Software Maintenance*. IEEE Press: Piscataway NJ, 2000; 131–142.
22. Aoki AK, Hayashi K, Kishida K, Nakakoji K, Nishinaka Y, Reeves B, Takashima A, Yamamoto Y. A case study of the evolution of JUN: An object-oriented open-source 3D multimedia library. *Proceedings IEEE 23rd International Conference on Software Engineering*. Computer Society Press: New York NY, 2001; 524–533.
23. Paulson JW, Succi G, Eberlein A. An empirical study of open-source and closed-source software products. *IEEE Transactions on Software Engineering* 2004; **30**(4):246–256.
24. Kemerer CF, Slaughter SA. An empirical approach to studying software evolution. *IEEE Transactions on Software Engineering* 1999; **25**(4):493–509.
25. Lehman MM, Ramil JF. The impact of feedback in the global software process. *The Journal of Systems and Software* 1999; **46**(2–3):123–134.
26. Greene WH. *Econometric Analysis* (5th edn). Prentice-Hall: Upper Saddle River NJ, 2003; 1026 pp.
27. Heales J. A model of factors affecting an information system's change in state. *Journal of Systems Management and Evolution* 2002; **14**(6):409–427.
28. Srinivasan K, Fisher D. Machine learning approaches to estimating software development effort. *IEEE Transactions on Software Engineering* 1995; **21**(2):126–137.
29. Lehman MM, Ramil JF. Rules and tools for software evolution planning and management. *Annals of Software Engineering* 2001; **11**(1):15–44.
30. McCabe TJ. A complexity measure. *IEEE Transactions on Software Engineering* 1976; **2**(4):308–320.
31. Halstead M. *Elements of Software Science*. Elsevier/North-Holland: New York NY, 1977; 127 pp.
32. Nikora AP, Munson JC. An approach to the measurement of software evolution. *Journal of Software Maintenance and Evolution: Research and Practice* 2005; **17**(1):65–91.
33. Low G, Leenanuraksa V. Software quality and CASE tools. *Proceedings of the Conference Software Technology and Engineering Practice (STEP '99)*. IEEE Press: Piscataway NJ, 1999.
34. Norman CR, Nunamaker JF. CASE productivity perceptions of software engineering professionals. *Communications of the ACM* 1989; **32**(9):1102–1108.
35. Necco CR, Tsai NW, Holgeson KW. Current usage of CASE software. *Journal of Systems Management* 1989; **40**(5):6–11.
36. Iivari J. Why are CASE tools not used? *Communications of the ACM* 1996; **39**(10):94–103.
37. Limayem M, Khalifa M, Chin WW. CASE tools usage and impact on system development performance. *Journal of Organizational Computing and Electronic Commerce* 2004; **14**(3):153–174.
38. Finlay PN, Mitchell AC. Perceptions of the benefits from introduction of CASE: An empirical study. *MIS Quarterly* 1994; **18**(4):353–370.

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

39. Banker RD, Kauffman RJ, Zweig D. Repository evaluation of software reuse. *IEEE Transactions on Software Engineering* 1993; **19**(4):379–389.
40. Norman RJ, Corbitt GF, Butler MC, McElroy DD. CASE technology transfer: A case study of unsuccessful change. *Journal of Systems Management* 1989; **40**(5):33–37.
41. Aiken L, West S. *Multiple Regression: Testing and Interpreting Interactions*. Sage Publications: Newbury Park CA, 1991; 212 pp.
42. Cohen J, Cohen P. *Applied Multiple Regression for the Behavioral Sciences*. Erlbaum: Hillsdale NJ, 1983; 545 pp.
43. Oldham G, Fried Y. Employee reactions to workplace characteristics. *Journal of Applied Psychology* 1987; **72**(1):75–80.
44. Huber PJ. The behavior of maximum likelihood estimates under non-standard conditions. *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*. University of California Press: Berkeley CA, 1967; 221–233.
45. White H. A heteroskedasticity-consistent covariance matrix estimator and a direct test for heteroskedasticity. *Econometrica* 1980; **48**(4):817–830.
46. White H. Maximum likelihood estimation of misspecified models. *Econometrica* 1982; **50**(1):1–25.
47. Belsley DA, Kuh E, Welsch RE. *Regression Diagnostics: Identifying Influential Data and Sources of Collinearity*. Wiley: New York NY, 1980; 285 pp.

AUTHORS' BIOGRAPHIES

Evelyn J. Barry received her doctorate and masters in Information Systems from the Tepper School of Business at Carnegie Mellon University. She also holds an MBA from the University of Wyoming and an MS in Mathematics from Colorado State University. She has worked in the information systems industry for over 20 years. Her research has been presented and published at the IEEE International Conference on Software Engineering, the IEEE International Conference on Software Maintenance, the Americas Conference on Information Systems, the International Conference on Information Systems and in *Management Science*, *Information Technology and Management* and *Empirical Software Engineering*.

Chris F. Kemerer is the David M. Roderick Professor of Information Systems at the Katz Graduate School of Business, University of Pittsburgh and an Adjunct Professor of Computer Science at Carnegie Mellon University. Previously, he was an Associate Professor at MIT's Sloan School of Management. He received the BS degree *magna cum laude* from the Wharton School at the University of Pennsylvania and the PhD degree from Carnegie Mellon University. He has served as an Associate Editor on eight editorial boards, is a past Departmental Editor for *Management Science*, and is the immediate past Editor-in-Chief of *Information Systems Research*.

Sandra A. Slaughter received her PhD in Management Information Systems from the University of Minnesota in 1995 and is currently an Associate Professor in the Tepper School of Business at Carnegie Mellon University (CMU). Prior to joining CMU, she worked in the software industry. Currently, she is conducting research funded by the National Science Foundation on software evolution and on project-management practices. She has published over 70 articles on software development issues in leading research journals, conference proceedings and edited books, and her work has received seven best paper awards. She also serves on editorial boards for major journals.

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/smr