On the Uniformity of Software Evolution Patterns

Evelyn J. Barry
Mays Business School
Texas A&M University
*ebarry@cgsb.tamu.edu****

Chris F. Kemerer
Katz Graduate
School of Business
University of Pittsburgh

Sandra A. Slaughter
Graduate School of
Industrial Administration
Carnegie Mellon University

Abstract

Preparations for Y2K reminded the software engineering community of the extent to which long-lived software systems are embedded in our daily environments. As systems are maintained and enhanced throughout their lifecycles they appear to follow generalized behaviors described by the laws of software evolution. Within this context, however, there is some question of how and why systems may evolve differently. The objective of this work is to answer the question, ***do systems follow a set of identifiable evolutionary patterns?*** *In this paper we characterize software evolution as software volatility, and examine the lifecycle evolution of 23 software systems. By using a series of mapping techniques and sequence analysis, we create volatility pattern vectors for each system. Factor analysis groups the 23 individual patterns into four clusters. Thus, we show by example that there are different patterns of system behavior within the context of software evolution.*

1. Introduction

Software engineers are well aware of the many legacy systems in production today. Even though the world didn't collapse under the threat of the Y2K bug, the adaptation of millions of lines of code emphasized the extent to which long-lived software systems are embedded in our daily business, educational and personal environments. Legacy systems often actively serve organizations far longer than the system designers and programmers who created them [1].

In point of fact, a truly successful system will continually be modified and enhanced [1]. Software lifecycle maintenance[1] is a forward-focused activity used to prolong the productive lifespan of a software system [1]. As much as 90% of lifecycle system costs are spent on software modifications after initial system implementation [2]. Therefore it is imperative that we understand the lifetime system transformations taking place. Such incremental modifications of software systems are often referred to collectively as *software evolution*. Formally, software evolution is defined as "…the dynamic behavior of programming systems as they are maintained and enhanced over their life times" [3].

Researchers have recognized the importance of software evolution for over three decades. Based on a series of empirical observations and analytical studies Lehman and colleagues have developed eight laws of software evolution. These laws describe the incremental transformations experienced by software systems during their lifecycles [4] [5]. Four of the eight laws describe changes in software products, and the remaining laws describe software processes. Empirical studies have provided further support for these characteristics of aging software system behaviors. [5-12].

The laws of software evolution describe general characteristics of the lifecycle transformations software systems experience. Within this context, however, there is some question of how and why systems may evolve differently [13]. Some systems are modified and stay productive for decades [14] while others are replaced in less than five years. Some systems require few changes and others need to be changed almost continuously. In earlier empirical studies of software evolution we have observed differences in patterns of software modifications suggesting that all systems do not follow identical patterns of software evolution [13]. Thus, while all software systems evolve, it is not clear how and why systems evolve in the way they do.

Our main objective in this work is to determine if distinct patterns of lifecycle software volatility can be identified, that is, ***do systems follow a set of identifiable evolutionary patterns?*** Using a series of mapping techniques and factor analysis we show by example that there is more than one pattern of software evolution. We use an inductive approach to develop a series of mapping techniques that allow us to condense calculated measures of software volatility. Sequential phase analysis techniques are then applied to identify patterns of software volatility among application software systems. Our empirical data are based on a 20-year log of lifecycle software maintenance for a large retailer [6]. The data represent over 25000 individual change events

[1] We include all software system modifications for corrective, adaptive and enhancement purposes as lifecycle maintenance.

in 3500 programs from 23 application systems. In the next section we apply a step-by-step methodology to map the lifecycle patterns of software evolution. These mapping techniques simplify measures of software volatility while retaining characteristic system behavior. We then use sequential phase analysis and non-parametric analyses to identify patterns of software evolution. In the final section of this paper we discuss contributions and implications of this work, and its significance to both researchers and practitioners. This work contributes to the set of methodologies we can use for describing, analyzing and comparing software evolution. By combining these techniques with cluster analyses we can detect similarities in system behaviors.

2. Methodology

In this section we demonstrate a progression of mapping techniques used to describe lifecycle software evolution patterns. This is an inductive process as we gradually apply a series of mapping techniques to provide insight into the nature of lifecycle software volatility. We first measure software volatility by calculating three dimensions of volatility: (1) amplitude, (2) periodicity and (3) deviation. We then classify the values of three dimensional software volatility into volatility classes and create lifecycle vectors of software volatility classifications. Using sequence (phasic) analysis, we identify distinct phases of volatility. Finally, we use factor analysis techniques to distinguish four distinct patterns of volatility. Our analysis is illustrated throughout using actual lifecycle software maintenance events for systems in the retailer's application portfolio.

2.1 Determining Software Volatility Dimensions

Analysis of lifecycle volatility begins with the calculation of software volatility measures. *Software volatility*, i.e. software change, is a multi-dimensional phenomenon. We describe software volatility with the dimensions of (1) *amplitude,* (2) *periodicity* and (3) *deviation* [15][2].

Amplitude measures the change in software system size. To allow comparison of different sized systems we normalize change in system size by the total system size.

Periodicity measures the mean time since software modification. For each program in a system we identify all software modifications completed during that time period and calculate the time since software modification (TSM) for each individual modification to the program. Software modifications are aggregated for the system and time period. The Mean time since software modification ($MTSM_{st}$) is calculated for each time period (*t*) in each system's (*s*) lifecycle. Periodicity is defined as MTSM normalized by system age at time period *t*. Periodicity suggests the frequency with which systems require change.

Deviation measures the variance of the TSM for system *s* and implemented during time period *t*. Just as periodicity is normalized by system age, deviation is normalized by system age squared at time *t*. A low deviation suggests that all programs in a system are changed on the same schedule. A high deviation indicates a wide disparity in TSM making it difficult for software managers to determine how many programs will need to be changed at time *t*.

We start by calculating amplitude, periodicity and deviation for each month in each system's lifecycle. This yields an unbalanced panel of 3201 system-month observations. If no activity occurs during a system-month observation, amplitude=0, periodicity = 1 and deviation=0, by definition. We illustrate these software volatility measures for systems 2 and 7 from our portfolio, and graph the results. (See Figures 1 and 2.)

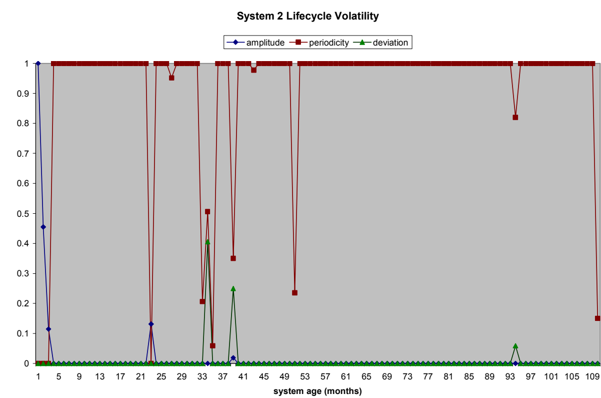

Figure 1: System 2 lifecycle volatility time series

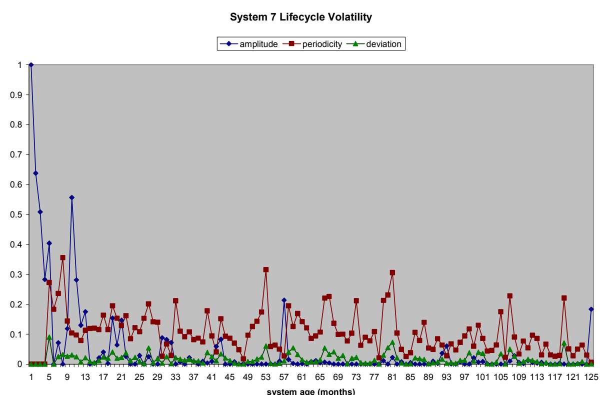

Figure 2: System 7 lifecycle volatility time series

[2] For a more formal explanation of each of these measures see the appendix.

Each of these graphs represents three time series of lifecycle volatility, i.e. amplitude, periodicity and deviation. However, with this level of detail it is difficult to discern patterns. The same calculations and detailed graphs were prepared for each of the remaining 21 software systems in our portfolio. No two systems display exactly the same volatility. The search for patterns in lifecycle software volatility may be obscured by 20 years of monthly calculations of the three dimensions of software volatility.

2.2 Classifying Software Volatility Dimensions

In order to help discern potential patterns in the data, we combine the three dimensions into a one-dimensional volatility classification for each of the system-month observations in our dataset. Each calculation of amplitude is marked as above average (= 1) or below average (= 0) amplitude.[3] The same procedure marks long (=1) or short (= 0) periodicity, and high (=1) or low (= 0) deviation. These binary indicators are combined to create eight triples signifying eight classifications of software volatility (A-H). (See Table 1.)

Table 1: Classifications of software volatility

Volatility Class	Amp.	Per.	Dev.	Description
A	Low	Long	Low	Least volatile: occasional small modifications occurring in a well-behaved pattern
B	Low	Long	High	Occasional small modifications with wide variance of behavior among system programs
C	Low	Short	Low	Constant small modifications occurring in a well-behaved pattern
D	Low	Short	High	Constant small modifications with wide variance of behavior among system programs
E	High	Long	Low	Occasional large modifications occurring in a well-behaved pattern
F	High	Long	High	Occasional large modifications with wide variance of behavior among system programs
G	High	Short	Low	Constant large modifications occurring in a well-behaved pattern
H	High	Short	High	Most volatile: constant large modifications with wide variance of behavior among system programs

[3] Average amplitude, average periodicity and average deviation are calculated as the mean values of amplitude, periodicity and deviation for the unbalanced panel dataset of 3201 system-month observations.

Volatility class A represents the volatility measure triple (low, long, low). Below average amplitude indicates either that there are no software modifications, or that all the modifications affect small amounts of code. Longer than average periodicity indicates that the programs being modified have not been modified for a very long time, if at all. Low deviation indicates that any programs being modified are all being changed at the same time with the same time interval since previous modification. We conclude that volatility class A has very few, if any, software modifications. At the other extreme, volatility class G has high amplitude, short periodicity and low deviation. This indicates that (1) software modification size is relatively large, (2) modifications occur frequently, and (3) the time since previous modification is consistent among the programs being changed. Volatility class G is the nightmare that software managers dread. All system programs require large modifications at each time internal.

The application portfolio contains the full lifecycle (maximum of 22 years) for each of the 23 application systems being analyzed, for a total of 3201 system-month observations. The number of occurrences for each volatility classification in our data set is listed in Table 2. There are very few occurrences of volatility classifications B, E, and F. Over half of the volatility occurrences are A or C, indicating smaller than average sizes and long periods between modifications. This indicates that the software portfolio, as a whole, is fairly stable.

Table 2: Frequency of volatility classes in software portfolio lifecycle

Volatility class	No. of Occurrences	Percentage of TOTAL
A	1398	43.67
B	12	0.37
C	973	30.40
D	448	14.00
E	1	0.03
F	1	0.03
G	281	8.78
H	87	2.72
TOTAL	3201	100.00

2.3 Describing Lifecycle Vectors of Software Volatility Classifications

A vector of volatility classifications can now be built to describe the lifecycle software volatility for each application system. Each element in the (N x 1) vector represents the software volatility classification for one month, i.e. N = number of months in system lifecycle. Software system 2 has a lifecycle of 110 months. Thus, its lifecycle volatility class vector has 110 elements.

Each element represents the volatility class for that month. In the same manner, we use the lifecycle volatility classifications for the 125 months of system 7's lifecycle to create a (125x1) lifecycle volatility vector.

The elements in each vector describe a chronological pattern of the software system's evolution. We use these lifecycle volatility vectors to complete our longitudinal analysis.

2.4 Identifying Phases of Lifecycle Software Volatility Using Phasic Analysis

There are two predominant methods for longitudinal analysis; time series analysis and phasic analysis [16]. Time series analysis is not appropriate given our use of nominal scale volatility classifications. Thus, we adopt a non-parametric sequential (phasic) analysis technique. This technique been applied in a wide variety of contexts, including the study of information system development processes [17]

Phasic analysis is based on three assumptions. First, events being analyzed can be assembled into groups of more than one occurrence. Second, events or activities cohere into larger events or patterns. Third, the structure of these events or patterns is the result of dynamics driving changes in behavior from one phase to the next [16]. In our use of this technique we look for phases of software volatility. We consider each month's volatility classification as a discrete event and use phasic analysis to identify consecutive months with similar lifecycle volatility classifications.

We use *Winphaser* as our phase sequence analyzer. Winphaser software was written by Michael Holmes, as adapted from [16]. Winphaser identifies phases of length n when it detects n consecutive like values. For sequences where no predominant classification can be identified Winphaser labels the phase as *pending* (p). The phase length parameter, n, gives Winphaser enough flexibility to allow its use in a variety of contexts. By changing the phase length researchers can simplify resulting sequential phase patterns. Smaller phase length parameters reduce the number of pending phases, but the resulting increased number of phases reduces our ability to see patterns or iterations of identified activities. However, longer phases provide a coarser description of behavioral patterns and may prevent a more meaningful description of volatility processes. We used bracketing to determine that a phase length equal to three (n=3) was the best tradeoff between these two extremes. Winphaser produced lifecycle phase maps for each of the 23 application systems in our software portfolio.

Winphaser uses Goodman and Kruskal's gamma, γ, to calculate the precedence scores of identified phases [18, 19]. The frequency of each phase is also reported in the gamma map. We used phase frequencies to calculate each phase type's proportion of total lifecycle phases. The precedence scores indicate the prevalent order of phase type occurrence. For example, suppose a system has two phase types reported, a and c. The frequencies for these phase types are 72 and 8, respectively. The precedence is reported as type a followed by type c. From this information we build a 10x1 volatility vector as follows (a,a,a,a,a,a,a,a,a,c), i.e. phase a followed by c in a 9:1 ratio.[4]

We use the gamma maps from Winphaser to create 23 (10x1) volatility patterns.

2.5 Converting a Nominal Scale of Volatility Classes to an Ordinal Scale of Volatility Levels

Volatility vectors using the nominal volatility classifications *A-H* and *P* (*pending*) are constructed for each of the application systems in our software portfolio. To further analyze the lifecycle volatility classifications listed in Table 1 we order the nominal volatility classifications according to level of volatility (low to high). There are so few occurrences of volatility classifications *B, E* and *F* in our data that phasic analysis failed to identify any phases of type *B, E* or *F*. Therefore, we order the remaining software volatility classifications (*A,C,D,G,H*). Based on the descriptions for each volatility class, we compare the level of volatility described by each triplet in Table 1. The remaining five volatility classes in order by level of volatility are (*A,D,C,H,G*). Phases labeled *pending,* volatility class *P*, occur when there is no single volatility class to identify three sequential entries. We insert volatility class *P* into the sequence to transition between classes with low amplitude and high amplitude. The ordering of volatility classes is shown in Table 3.

Table 3: Ordering of volatility classifications

Nominal scale	A	D	C	P	H	G
Order	1	2	3	4	5	6

2.6 Grouping Systems with Similar Lifecycle Volatility Vectors

Lifecycle volatility vectors are constructed using the ordered volatility classifications. We now have a set of lifecycle vectors describing the volatility patterns for each system. The ordered volatility vectors have volatility levels ranging 1 to 6, and each element of the vector represents 10% of the phases in the system's lifecycle. The volatility vectors can be used to compare and group systems with similar patterns of behavior.

[4] We are normalizing all vectors and using pro rata share for each volatility class. This helps us compare patterns of system behavior across systems of different ages.

The volatility vectors represent phase sequences that may occur iteratively throughout a system's productive life span. A *base system* was selected from each group to display the main pattern of behavior for the group. Groups 1-4 are numbered in order by overall volatility, lowest to highest.

We used several non-parametric analyses to verify our specification of volatility vector groups. We confirmed our selection by calculating the difference of each cluster member system from the representative base member system. We calculate the distance from one vector to another as the difference between each element in one vector from the corresponding element in another vector. The sum of the differences can be used as a measure of the distance (or similarity) between two vectors.

A volatility vector for a completely stable system would be a 10x1 volatility vector of (1,1,1,1,1,1,1,1,1,1). The volatility vector for system 2 is (1, 1, 1, 1, 1, 1, 1, 1, 1, 6). The distance between these vectors can be calculated as the sum of the differences of each pair of corresponding elements[5]. The distance between system 2 and complete stability is (0+0+0+0+0+0+0+0+0+5) = 5. The volatility vector for system 7 is (5, 4, 2, 2, 2, 2, 2, 3, 3, 3). The distance between the system 7 and complete stability is (4+3+1+1+1+1+1+2+2+2) = 18. With this method we can see that system 7 is more volatile than system 2. In Table 4 we list each of the 23 systems with its ordinal volatility vector and its distance from a vector for a completely stable system. We visually arranged the volatility vectors in four groups by their similarities. Group numbers are assigned in order by overall volatility. We selected representative system vectors as base systems for each group are indicated by an '*'.

2.7 Using Cluster Analysis to Confirm Groupings of Lifecycle Volatility Vectors

To further verify our selection of these groups we created a file of 23 records, each containing the lifecycle volatility vector found in Table 4. To confirm our manual arrangement of each of the four groups we applied hierarchical cluster analysis using linkage between group clusters[6] with interval squared Euclidean distance measures (SPSS V10.1). Our visual grouping of the volatility patterns aligns with the grouping yielded by the hierarchical cluster analysis. Obtaining the same results by two different methodologies provides support for the strength of the groupings.

[5] In this calculation to make the analysis tractable the ordinal scale of volatility 1-6 is assumed to serve as an interval scale as well.

[6] Average linkage between group clusters maximizes the distance between 2 clusters where distance is the average of distances of all pairs of cases where one member of the pair is from one of the clusters.

Table 4: System volatility vectors by group number

Pattern Group	Sys-tem #	Ordered Volatility Vector										Distance to stability
1	23*	1	1	1	1	1	1	1	1	1	1	0
1	19	1	1	1	1	1	1	1	1	1	1	0
1	14	4	1	1	1	1	1	1	1	1	1	3
1	2	1	1	1	1	1	1	1	1	1	6	5
1	16	1	1	1	1	1	1	1	1	1	4	3
2	13*	1	1	1	1	1	1	6	3	3	3	11
2	17	1	1	1	1	1	1	6	3	3	3	11
2	9	1	1	1	1	1	3	3	3	2	2	8
2	15	1	1	1	1	1	4	3	3	3	3	11
2	21	1	1	1	1	1	1	4	4	4	4	12
2	18	1	1	1	1	1	4	4	3	3	3	12
3	12*	1	1	1	4	4	3	3	3	3	3	16
3	1	1	4	4	4	2	2	2	2	3	3	17
3	5	1	6	3	3	3	4	2	2	2	2	18
3	8	1	1	6	3	3	3	3	3	3	3	19
3	6	2	2	2	3	3	3	3	3	3	3	17
3	20	2	4	4	4	4	3	3	3	3	3	23
3	3	1	1	1	6	4	3	3	3	3	2	17
4	7*	5	4	2	2	2	2	2	3	3	3	18
4	10	5	4	2	2	2	2	2	3	3	3	18
4	4	4	4	1	1	1	1	1	1	3	3	10
4	22	6	4	4	4	4	2	2	3	3	3	25
4	11	6	2	4	4	4	4	3	3	3	3	26

3. Discussion

Our objective in this research was to determine whether all software systems traverse the same lifecycle evolution pattern or whether there are distinctly different patterns among software systems. We used lifecycle volatility as representative of the dynamic behavior of software systems, i.e. software evolution.

We find that systems do not all follow the same volatility pattern, and that there are distinct pattern groups of lifecycle volatility. Using a progression of mapping techniques we simplified the three-dimensional time series of software volatility to an Nx1 vector of software volatility classifications. The end result of these mappings displays a volatility pattern of software system behavior. We prepared volatility patterns for each of the 23 application systems in our software portfolio.

We found four distinct groups of volatility patterns for the systems in our software portfolio. Each group is represented by a base system. Group 1 contains those systems whose volatility is relatively low, i.e., the systems are stable, for at least 90% of the time. The

pattern for group 2 is stable for the first 50-60% of the pattern followed by higher volatility in 40-50% of the pattern cycle. Group 3 volatility patterns are stable for a relatively short time, i.e. low volatility for the first 10% of the cycle. This brief level of stability is followed by higher volatility for the remaining cycle pattern. Group 4 volatility patterns start with high volatility, and volatility remains relatively high for the entire cycle pattern.

Our analysis of lifecycle software volatility uses a three-dimensional measure of software volatility: amplitude, periodicity and deviation. A fourth dimension, time, is added when we analyze changes in software volatility throughout the productive lifecycle of a software system. Our choice of mapping techniques allowed us to transform and simplify a 4-dimensional phenomenon into a 2-dimensional one. This process clarified the views of 23 software system behaviors and allowed us to perceive similarities of behaviors among the systems in our software portfolio.

Our software portfolio generated a set of four distinct volatility patterns. Depending on the age of a system the pattern may represent a full lifecycle from implementation to present, or the pattern may occur repeatedly in a system's lifecycle.

Two systems may have identical volatility patterns but different lifecycle volatility. For example, systems 7 and 10 have identical lifecycle volatility patterns but dissimilar volatility time series. Lifecycle volatility time series graphs for systems 7 and 10 are displayed as figures 2 and 3, respectively. The volatility vector (pattern) for system 7 is identical to that for system 10. (See Table 4.) However, when you compare the lifecycle volatility time series graphs (Figures 2 and 3) you see that systems 7 and 10 have the same proportion of low volatility observations indicated by longer than average periodicity. The smoothing effect of assigning volatility classes eliminated some of the fog and helped identify similarities of behavior.

Figure 3: System 10 lifecycle volatility

By using the ordinal scale of software volatility (Table 3), we can plot lifecycle volatility from the phase maps of systems 7 and 10. (See Figures 4 and 5.) Even though systems have identical volatility patterns, there are differences in their lifecycle volatility. One system is much older than the other and exhibits more iterations of the volatility pattern. If managers know the pattern of behavior for a given system they may be able to improve their prediction of resources needed for future lifecycle maintenance support of that system.

Figure 4: System 7 lifecycle volatility (ordinal scale)

Figure 5: System 10 lifecycle volatility (ordinal scale)

4. Conclusion

Software evolution has been researched for over three decades. While we recognize the many universal elements of software evolution we still observe a wide

variety of system behaviors over time. In this work we measure this dynamic behavior as software volatility.

Software volatility is a complex phenomenon. We use a three-dimensional measure of volatility to capture the amplitude (size), periodicity (frequency) and deviation (consistency) of software system change. We add another dimension, time, by calculating software volatility for each time period (e.g. each month) in a software system's complete lifecycle. Using a series of mapping techniques we converted the three-dimensional measures of software volatility to a nominal classification (A-H) describing volatility in more general terms. A chronological Nx1 vector was created for each system. The vector had N entries, one for each time period (month) in the system's lifecycle. The vector was processed using phase sequence analysis software tool (*Winphaser*). The resulting gamma analysis allowed us to create a 10x1 volatility vector for each software system showing the sequence and proportion of each type volatility experienced by the system. These vectors were then ordered by overall level of volatility.

The individual volatility patterns for the systems in our software portfolio were grouped for similarity by visually identifying four distinct groups of volatility patterns. As a robustness check we used hierarchical clustering, and found the same four clusters of volatility patterns across the system portfolio.

This work contributes to the set of methodologies we can use for describing, analyzing and comparing the evolution of individual software systems. By combining these techniques with cluster analyses we can detect similarities in system behaviors.

This method enables us to distinguish at least 4 main groups of volatility patterns in our software portfolio. Even though there are universal characteristics of software evolution, there are distinct differences in patterns of system lifecycle behavior. From the laws of software evolution we know that software change is inevitable. From our previous work we know how to measure it. From this work we can describe and distinguish differences in software volatility patterns among systems.

The volatility pattern groups provide a separate and distinct classification of software systems. Now we need to determine what these pattern groups can tell us about other similarities among systems. Do certain pattern groups serve particular types of users? We ranked the groups from 1 to 4 by their overall level of volatility. Does it necessarily follow that group 1 will have lower lifecycle costs than groups 2, 3 and 4? Is it true that group 4 will be the most expensive? What do we know *a priori* about a system that will help us to gauge full lifetime costs of a software system being considered for purchase? Can we use pattern group as a factor in the repair/replace decision? Future research can draw upon the results presented here to answer these questions and many others. This analysis has added another perspective of software volatility and improved our general understanding of software evolution and its effect on lifecycle software maintenance outcomes.

5. Appendix

5.1 Amplitude

Amplitude measures the size of software modifications. Nearly two decades ago researchers began measuring software size. Amplitude can be measured as the sum of the size of all modifications made to a software system during time period *t*.

$$Amplitude = [\sum_{all\ modifications} size(modification_i)]_t / [total\ system\ size]_t$$

We can use any of the measurement units for measuring modification size. Division of the sum of modification sizes by the total size of the system creates a bounded scale invariant measure. Amplitude is an aggregate measure calculated for each of the 3201 system-month observations.

5.2 Periodicity

Periodicity calculation starts with measurement of time between software modifications, TSM_t, for any program modified during time period *t*. We use all software system modifications regardless of their purpose, e.g. corrective, adaptive or enhancement. Time between software modification, TSM, is the elapsed time since the previous modification.

TSM_{jt} = *TSM for change event j in time period t.*[7]

Mean time between software modification, $MTSM_t$, is the mean TSM_j for a system during time period *t*. [8]

$$MTSM_t = (1/N_t)\ \Sigma \sum_{j=1}^{N_t} TSM_j$$

Where N_t = *total number of change events for the system during time period t.*

$Periodicity_t$ is $MTSM_t$ normalized by system age at time *t,* i.e.

$$Periodicity_t = MTSM_t / t.$$

[7] Note: i = 0 is the point in time when the software system is initially implemented. Thus, at any time i is equal to the system age.

[8] A change event is a recorded software modification or program creation.

5.3 Deviation

Deviation describes how far system behavior varies from the pattern defined by amplitude and periodicity. By definition, deviation will vary over a software system's lifecycle, but not within each time period *t*. TSM_{jt} will vary within each time period. We could use variance of the TSM_{jt}, variance(TSM_{jt}), as our measure of deviation if it were bounded and scale invariant. Scale invariance can be obtained by calculating the variance of normalized TSM_{jt}, i.e. $NTSM_{jt} = (TSM_{jt})/t$. We refer to the normalized variance as $Deviation_t$.

$$Deviation_t = variance(NTSM_t) = variance(TSM_{jt}/t) = (1/t^2)[variance(TSM_{jt})]$$

6. References

[1] E. B. Swanson and E. Dans, "System Life Expectancy and the Maintenance Effort: Exploring Their Equilibration," *MIS Quarterly*, vol. 24, pp. 277-297, 2000.

[2] K. H. Bennett, "Software Evolution: past, present and future," *Information and Software Technology*, vol. 39, pp. 673-680, 1996.

[3] L. A. Belady and M. M. Lehman, "A Model of Large Program Development," *IBM Systems Journal*, vol. 3, pp. 225-252, 1976.

[4] M. M. Lehman and L. A. Belady, *Program Evolution: Processes of Software Change*. London: Academic Press, 1985.

[5] M. M. Lehman, J. F. Ramil, P. D. Wernick, P. D.E., and W. M. Turski, "Metrics and Laws of Software Evolution - The Nineties View," presented at Metrics '97, the Fourth International Software Metrics Symposium, Albuquerque NM, 1997.

[6] C. F. Kemerer and S. A. Slaughter, "An Empirical Approach to Studying Software Evolution," *IEEE Transactions on Software Engineering*, vol. 25, pp. 493-509, 1999.

[7] C. F. Kemerer and S. A. Slaughter, "A Longitudinal Analysis of Software Maintenance Patterns," presented at ICIS International Conference on Information Systems, Atlanta, GA, 1997.

[8] V. Basili, L. Briand, S. Condon, Y. K. Kim, and W. V. Melo, J., "Understanding and Predicting the Process of Software Maintenance Releases," presented at ICSE 18th International Conference on Software Engineering, Berlin, Germany, 1996.

[9] D. Gefen and S. L. Schneberger, "The Non-Homogeneous Maintenance Periods: A Case Study of Software Modifications," presented at ICSM IEEE Conference on Software Maintenance, Monterey CA, 1996.

[10] C. R. Cook and A. Roesch, "Real-Time Software Metrics," *Journal of Systems and Software*, vol. 24, pp. 223-237, 1994.

[11] C. K. S. Chong Hok Yuen, "An Empirical Approach to the Study of Errors n Large Software Under Maintenance," presented at ICSM 2ns IEEE Conference on Software Maintenance, Washington, D.C., 1985.

[12] C. K. S. Chong Hok Yuen, "A Statistical Rationale for Evolution Dynamics Concepts," presented at ICSM IEEE Conference on Software Maintenance, Austin, TX, 1987.

[13] C. F. Kemerer and S. A. Slaughter, "Determinants of Software Maintenance Profiles: An Empirical Investigation," *Journal of Software Maintenance*, vol. 9, pp. 235-251, 1997.

[14] R. Kalakota and A. B. Whinston, *Electronic Commerce: A Manager's Guide*. Reading, MA: Addison-Wesley, 1996.

[15] E. J. Barry and S. A. Slaughter, "Measuring Software Volatility: A Multi-Dimensional Approach"," presented at ICIS International Conference on Information Systems, Brisbane, Australia, 2000.

[16] M. E. Holmes and M. S. Poole, "Longitudinal Analysis," in *Studying Interpersonal Interaction*, S. Duck, Ed. New York: The Guilford Press, 1991, pp. 286-302.

[17] N. F. Sabherwal and D. Robey, "An Empirical Taxonomy of Implementation Processes Based on Sequences of Events in Information System Development," *Organization Science*, vol. 4, pp. 548-576, 1993.

[18] S. Siegel and N. J. Castellan, Jr., *Nonparametric Statistics for the Behavioral Sciences*, second ed. New York: McGraw-Hill, 1988.

[19] D. C. Pelz, "Innovation Complexity and the Sequence of Innovating Stages," *Knowledge, Creation, Diffusion Utilization*, vol. 6, pp. 261-291, 1985.

*** 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, Carnegie Mellon University, [NSF-IGERT], and the Institute for Industrial Competitiveness, University of Pittsburgh, Katz Graduate School of Business.