EVALUATION AND STUDY OF SOFTWARE DEGRADATION IN THE EVOLUTION OF SIX VERSIONS OF STABLE AND MATURED OPEN SOURCE SOFTWARE FRAMEWORK

Sayyed Garba Maisikeli

College of Computer and Information Sciences
Al-Imam Muhammad Ibn Saud Islamic University
Riyadh, Kingdom of Saudi Arabia
maisikel@ccis.imamu.edu.sa

ABSTRACT

When a software system evolves, new requirements may be added, existing functionalities modified, or some structural change introduced. During such evolution, disorder may be introduced, complexity increased or unintended consequences introduced, producing ripple-effect across the system. JHotDraw (JHD), a well-tested and widely used open source Java-based graphics framework developed with the best software engineering practice was selected as a test suite. Six versions were profiled and data collected dynamically, from which two metrics were derived namely entropy and software maturity index. These metrics were used to investigate degradation as the software transitions from one version to another. This study observed that entropy tends to decrease as the software evolves. It was also found that a software product attains its lowest decrease in entropy at the turning point where its highest maturity index is attained, implying a possible correlation between the point of lowest decrease in entropy and software maturity index.

KEYWORDS

Software Evolution, Software maintainability and degradation, Change ripple-effect, Change Impact, Change Propagation

1. INTRODUCTION

After a software system is developed, there is a high possibility that it may undergo some evolution due to change in business dynamics, response to environmental change, bug fix exercise, improving design, preventive maintenance or intentional modifications for overall improvement of the performance of the software system. A small change in an object-oriented software system however, may produce major local and nonlocal ripple effects across the software system. The goal of software evolution is to explore and study ripple-effects and cumulative effects of changes over time; observing whether quality, stability and extendability of the software are affected as the software system evolves from one version to another. Considering the size and complexity of the modern software systems, tracking and discovering parts of the software impacted, risks associated with change, and consequences of a change cannot be overemphasized.

When used properly, change impact assessment can help in managing and assessing software maintenance risks, thereby providing guidelines for effective software evolution implementation. This project investigated six versions of JHotDraw, a widely used open-source Java graphics framework as a test suite. The novel idea about this project is that while similar research efforts used static data collection methods, this project applied dynamic data collection methods on the test suite software under study.

According to [1], software maintenance includes corrective, adaptive and perfective maintenance enhancements which are technically not a part of software maintenance but, being a post-release activity.

Identifying potential consequences of a change or estimating what needs to be modified to accomplish a change may be a daunting task. According to [2] when a software system undergoes modifications, enhancements and continuous change, the complexity of software system eventually increases, with a possibility that some level of disorder may be introduced, making the software system becoming disorganized as it grows, thereby losing its original design structure. Considering the size and complexity of the modern software systems, tracking the effect of the change, understanding change impact and what parts of the software are affected and possible risks associated with a proposed change and potential consequences (side-effects) of a change cannot be overemphasized. When used properly and effectively, software change impact assessment can proactively provide a means of managing software maintenance risks and help guide the implementation of the software change.

On the issue of measuring software degradation, [3 and 4] suggest the use of entropy as an effective measure, and opined that software declines in quality, maintainability, and understandability as it goes through its lifetime. This paper sets out to study six consecutive versions of JHotDraw, a matured and well-structured open source graphics software framework that has been widely used in many research projects as test subject software. Each of the test versions was subjected to dynamic profiling and tracing routine that collected data from which Shannon entropy and software maturity index were derived.

The goal was to observe the entropy level change, and whether there is any correlation between entropy and software maturity index as the software system evolves from one version to another.

2. RELEVANCE

According to [5], the two most common meanings of software maintenance include defect repairs and enhancements or adding new features to existing software applications. Another view expressed by [5] also opined that the word “maintenance” is surprisingly ambiguous in a software context and that in normal usage it can span some twenty-one forms of modification to existing applications.

According to [6], almost 50% of software life cycle cost is attributed to maintenance; and yet, relatively very little is known about the software maintenance process and the factors that influence its cost. Considering the cost magnitude associated with maintenance and the ever-increasing size and sophistication of modern day software systems; it is then clear that software maintenance cost decisions and associated evolution risks cannot be taken lightly.

3. RELATED STUDIES

In a software evolution research, [7], analyzed change of software complexity and size during software evolution process, and discussed the characteristics related to the Lehman's Second Law (Lehman et al., 1997), which deals with complexity in the evolution of large software systems and suggests the need for reducing complexity that increases, as new features are added to the system during maintenance activities. Also, [7] opined that addition of features leads to the change of basic software characteristics (such as complexity/entropy) in the system. Their paper used this change as a means to determine different stages of evolution of a software system, proposing a software evolution visualization method called Evolution curve (or E-curve).

Discussing software maintenance consequences, [5] also observed that in every industry, maintenance tends to require more personnel than those building new products. For the software industry, the number of personnel required to perform maintenance is unusually large and may top 75% of all technical software workers. The main reasons for the high maintenance efforts in the software industry are the intrinsic difficulties of working with aging software, and the growing impact of mass updates. In an empirical study conducted by [8], thirteen versions of JHotDraw and 16 versions of Rhino released over the period of ten years were studied, where Object-Oriented metrics were measured and analyzed. The observed changes and the applicability of Lehman’s Laws of Software Evolution on Object Oriented software systems were tested and compared.

In a research paper, [9] presented how graph-based characterization can be used to capture software system evolution and facilitate development that helps estimate bug severity, prioritize refactoring efforts, and predict defect-prone releases. Also, [10] presented a set of approaches to address some problems in high-confidence software evolution. In particular, a history-based matching approach was presented to identify a set of transformation rules between different APIs to support framework evolution, and a transformation language to support automatic transformation.

In another paper, [11] compared software evolution to other kinds of evolutions from science and social sciences, and examined the forces that shape change, and discussed the changing nature of software in general as it relates to evolution, and proposed open challenges and future directions for software evolution research. From software evolution point of view, [12] described how and when the software evolution laws, and the software evolution field, evolved, and also addressed the current state of affairs about the validity of the laws, how they are perceived by the research community and the developments and challenges that are likely to occur in the coming years.

In contrast, this paper focuses on measuring software degradation in the evolution of six versions of a large-scale open-source software system with a special focus on investigating the introduction of disorder and observing the software maturity level as the software system evolves from one version to another.

4. METHODOLOGY

In order to explore and investigate the effect of change and its impact on the amount of disorder introduced as a software system evolves from one version to another, this study considers six versions of JHotDraw (JHD) as a test suite. These six versions were produced in a period of about five years (2006 to 2011), reflecting its natural evolution as new requirements were added, existing functionalities modified or enhanced, and some were deleted.

4.1 Test Program (JHotDraw)

JHotDraw is a very popular, mature and well documented widely used open-source Java-based graphics framework that has been used extensively in many software engineering research projects as a test suite. This framework provides a skeleton for developing highly structured drawing editors and production of document-oriented applications. The framework is known to be heavily loaded with numerous design patterns, developed based on the solid object-oriented principles, and based on the best software engineering practices.

To justify using the six different versions of JHotDraw in this research, we referred to some authors who have used them previously; this includes [7] and [8] where they recommended the use of JHotDraw as an Aspect Mining validation benchmark. Also, [13] and [14] used JHotDraw as a benchmark test suite in their research work. In addition, [8] used JHD as one of the test suites in his project.

Since JHotDraw is a mature and widely used test software programs, this research project also adopted it as a test program. It should be noted that, although there are ten documented versions of JHotDraw, seven versions are considered in this research study because the difference between earlier versions (7.0.6 and 7.07) is minimal as explained by [7]. To help us understand the chronological nature of the test program and its various versions, some characteristics details are presented in table 1 below:

Table I. Characteristics of the six versions of JHotDraw

Versions	Release Date	Size (MB)	LOC	No. Classes	NOM	No. of Attributes
Version 7.0.9	6/21/2007	11.2	52,913	487	4,234	1090
Version 7.1	3/8/2008	27.6	53,753	485	2,800	1087
Version 7.2	5/9/2008	22.6	71,675	621	5,486	1479
Version 7.3.1	10/18/2009	22.7	73,361	638	5,627	1516
Version 7.4.1	1/16/2010	22.6	72,933	639	5,582	1455
Version 7.5.1	1/8/2010	23.3	79,275	669	5,845	1599
Version 7.6	6/1/2011	23.5	80,169	672	5,885	1606

Seven different versions of JHotDraw are evaluated and tested (see table 1). Each of the versions of JHotDraw) were dynamically profiled and traced through the use of AspectJ run-timed weaver. (AspectJ runtime weaver is discussed in section 4.2). In order to maximize code coverage, forty-six of the major functionalities of each of the JHD applet versions were exercised as they execute. The granularity level adopted in targeting the various test program artifacts for data collection in this project is at the method level, rather than at class level.

One of the reasons for the choice is that methods in Object Oriented programming represent a modular unit by which programmers attribute well-defined abstraction of ideas and concepts. [15], defined methods in object-oriented paradigm as self-contained units where distinct tasks are defined, and where implementation details reside, making software reusability possible. According to [16], methods are less complex than classes, are easier to compare, and provide significant coverage and easy distinction, and have a high probability of informal reuse. [17] Observed that all known dynamic Aspect Mining techniques are structural and behavioral and work at method granularity level.

Event traces were dynamically collected as the test software versions were executed, with the AspectJ runtime weaver seamlessly running in the background. The runtime weaver has the capability to dynamically insert probes at selected points in the target test software (in this case class methods) at specify points known as (joinpoints), where all method executions were traced and data collected. In this project, we are interested in the sequence and frequency of calls, rather than method fan-in and fan-out. Frequency counts for each method calls were tallied, from which probabilities of method invocation were calculated and assigned.

Note that, since methods with the same name in different classes may be counted as one and the same, we left the class prefix along with method names to make sure that such methods are counted distinctly and correctly. Note also that duplicate method calls were left intact in the data collected, since removing such duplicate calls will distort the frequency counts of the method invocations.

The assigned probabilities represent the probability that such code units will be invoked as the system is run. It is from this frequency count that the entropy is calculated as the software changes from one version to another. The other metric used was software maturity index (SMI); this was derived from the static data collected from documentations produced by [15]. Explanation on how these two metrics are used are discussed in the next few pages.

4.2 Dynamic Data Collection tool (AspectJ Weaver)

AspectJ runtime weaver allows probes to be inserted at specific points of interest statically or dynamically when the software source code to be profiled executes. Code that allows observing tracing or changing the software source code is weaved according to the required action specified in what is called (pointcut). The weaved/inserted code logs the behaviors of the test software, track its actions based on the given behavior specified by pointcut; in our case, tracing and profiling each of the methods in our test software system as they are executed or invoked. AspectJ runtime weaver can be used to seamlessly and dynamically collect data on the test software as it executes.

The weaver evaluates the pointcut expressions and determines the (joinpoints) where code from the aspects is added. This may happen dynamically at runtime or statically at compile time. The runtime weaver then creates a combined source by weaving the source code of the aspects into the sources of the program under investigation. The generated program code is then compiled with the compiler of the component language, which is Java in our case.

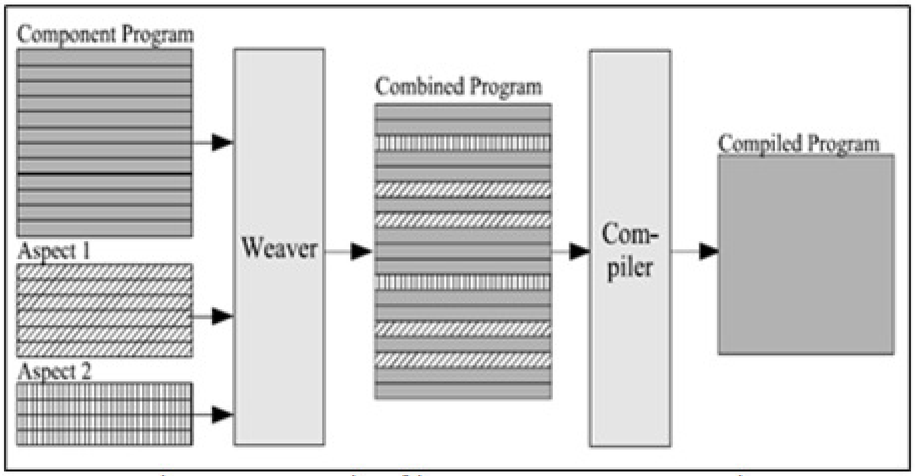

Figure 1. Example of how AspectJ Weaver works

4.3 Metrics derived from collected data

To assess, evaluate and study the nature of the test software as it evolves from one version to another, two software metrics were considered in this research project. Included are the Shannon's Entropy and Software maturity Index (SMI). These metrics were derived from the datum collected as the test programs run.

4.3.1 Shannon's Entropy

Within the context of software evolution, entropy can be thought of as the tendency for a software system that undergoes continuous change eventually become more complex and disorganized as it grows over time, thereby becoming more difficult and costly to maintain.

One of the metrics derived in this project is Entropy, with this metric; we will be able to find a way to assess whether the test software versions get degraded as they evolve from one version to another. According to [4], when investigating and studying the effect of a change in a software system, Shannon's equation may be better than complexity averaging. According to [1], in addition to measuring disorder introduced into software evolution, entropy also provides a measure of the complexity of the software system. [3], [20] stated that entropy can anecdotally be defined to mean that software declines in quality, maintainability, and understandability through its lifetime. For effective measurement and assessment of software degradation, [4] recommended the use of entropy for the study of software degradation.

Many variations of Shannon's entropy formula is presented in academic papers, but the generalized Shannon's entropy formula is expected as follows :

$$H_I = -\sum_i p_i \ln p_i \, .$$

Where

H = System Complexity Entropy,

p_i = Probability that method m_i in test software is invoked
i = Integer value *1, 2…j*, representing each of the categories considered

Note that the negative sign in the equation is introduced to cancel the negative sign induced by taking the log of a number less than 1.

As explained earlier in the introduction section, the entropy probability in this project is derived based on the method invocation frequency counts collected when the different versions of the test programs are executed and exercised. As an example of how entropy is derived in this project, consider the example of a software system **S** with three classes **C**$_1$, **C**$_2$, **C**$_3$. Methods (**m**$_{11}$, **m**$_{21}$) are contained in **C**$_1$, methods (**m**$_{12}$, **m**$_{22}$, **m**$_{32}$) contained in **C**$_2$, and (**m**$_{13}$, **m**$_{23}$, **m**$_{33}$, **m**$_{43}$, **m**$_{53}$) contained in **C**$_3$. The numbers shown beside class methods are representations of the frequency of method invocations when the test software was exercised.

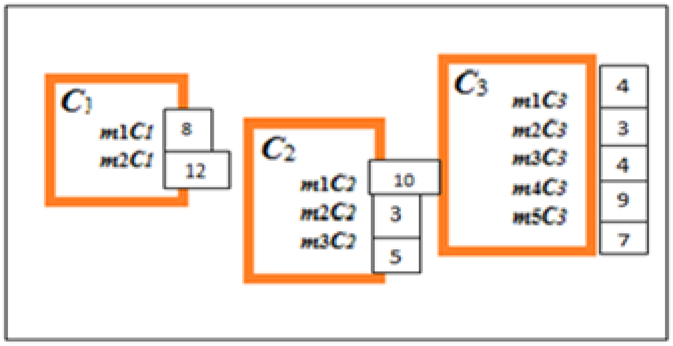

Figure 2. Example of method invocation from three different classes in (software **S**)

Based on the given example of the three classes and the associated method invocations shown in figure 2 above, we can construct probability required for the calculation of the entropies for all methods in the software being tested as shown in table 2 below

Table II. Example of calculation of probability of method invocation.

Classes	**Invoked Methods**	**Invocation Frequency**	**Invocation Probability**
C_1	m_1C_1	8	0.1231
	m_2C_1	12	0.1846
C_2	m_1C_2	10	0.1538
	m_2C_2	3	0.0462
	m_3C_2	5	0.0765
C_3	m_1C_3	4	0.0615
	m_2C_3	3	0.0462
	m_3C_3	4	0.0615
	m_4C_3	9	0.1385
	m_5C_3	7	0.1077
		Total 65	0.9696

Figure 3 below shows a graph of chronological change of JHD entropy values from one version to another. To construct the graphs displayed in figure 3, entropy calculated for a version was compared to the previous one. As depicted, it should be noted that initially, the entropy remains stubbornly the same, but at a later stage, the entropy dropped consistently as the test software versions transition from one version to another.

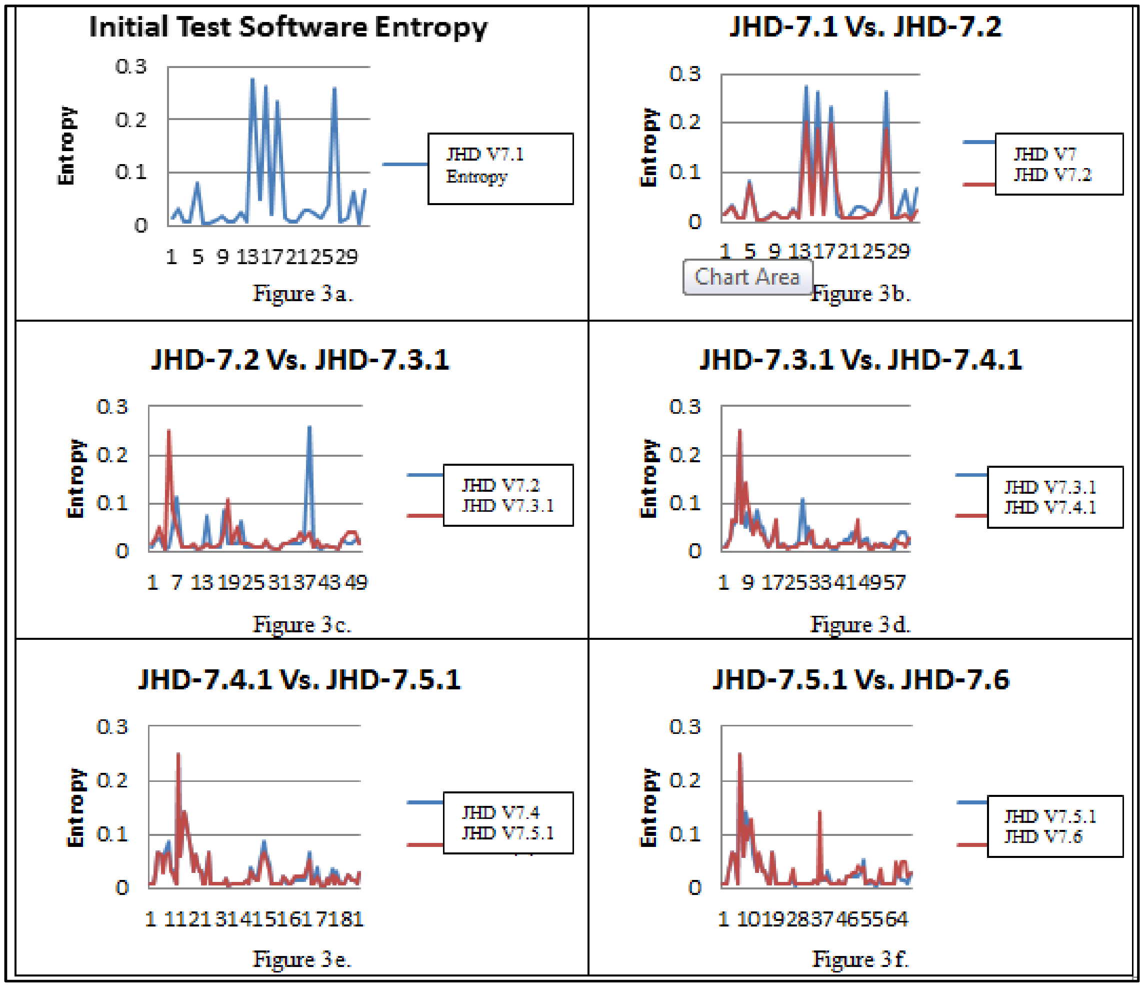

Figure 3 Entropy graph Version to Version

The graph shown in figure 3a is for the initial version of JHD (version 7.0.1) before any change is made. The subsequent figures (3b through 3f) are a superimposition of entropy values representing transitions from one version to another (two versions at a time). From these graphs, a gradual decrease in entropy values can be observed. The high spikes in the middle of each graph are indications of changes reused packets and other add-in modules have undergone throughout the transitional evolution of the test software system.

4.3.2 Software Maturity Index (SMI)

When discussing software maturity, [6] defined Software Maturity Index (SMI) as a metric that provides an indication of the stability of a software product (based on changes that occur for each release of the product). The software maturity index is computed in the following manner:

$SMI = [M_T - (F_a + F_c + F_d)]/M_T$

Where,

M_T= number of modules in the current release
F_c= number of modules in the current release that have been changed
F_a= number of modules in the current release that have been added
F_d= number of modules from the preceding release that were deleted in current release

Software maturity index (SMI) is especially used for assessing release readiness when changes, additions or deletions are made to an existing software system. An observation made by [6] emphasized that, as SMI approaches 1.0, the product begins to stabilize. SMI may also be used as metric for planning software maintenance activities. The mean time to produce a release of a software product can be correlated with SMI, and empirical models for maintenance effort can be developed. In this project, this metric was derived from the chronology of JHotDraw Updates/Additions/Deletions documented and presented by [5]. In this project, the calculation of SMI is based on the package rather than at class or method granularity levels.

Table 3. Data for Software maturity index calculation

From Version to Version	**No. Of Packages**	**Packages Added**	**Packages Changed**	**Packages Deleted**	**Calculated (SMI)**
JHD-V7.1 to JHS-V7.2	46	8	24	0	0.30
JHD-V7.2 to JHS-V7.3.1	46	0	23	0	0.50
JHD-V7.3.1 to JHS-V7.4.1	44	6	0	2	0.81
JHD-V7.4.1 to JHS-V7.5.1	46	3	6	0	0.80
JHD-V7.5.1 to JHS-V7.6	45	1	7	1	0.80

From archive data obtained from [18] and [19], a summary of all addition, changes, and deletions made to JHD versions 7.1 through version 7.6 were used to calculate the software maturity index as shown in table 3 above. Also, from this data, the SMI graph is drawn and displayed in figure 4 below. From this graph, it will be seen that the Maturity Index (MI) increases and then levels off as the optimal level of 0.8 is reached, starting from the evolution transition point (V7.3.1 to V7.4.4), stagnating all the way through (V7.6).

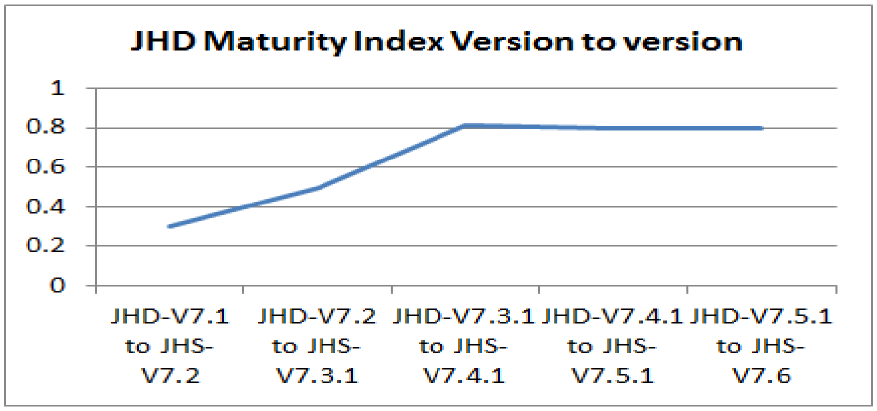

Figure 4 Inter-version Maturity Index

To further view the nature of the JHD evolution and the attained maturity pictorially, the SMI is calculated from the collected transition data for all versions and graphed as shown in figure 5 below.

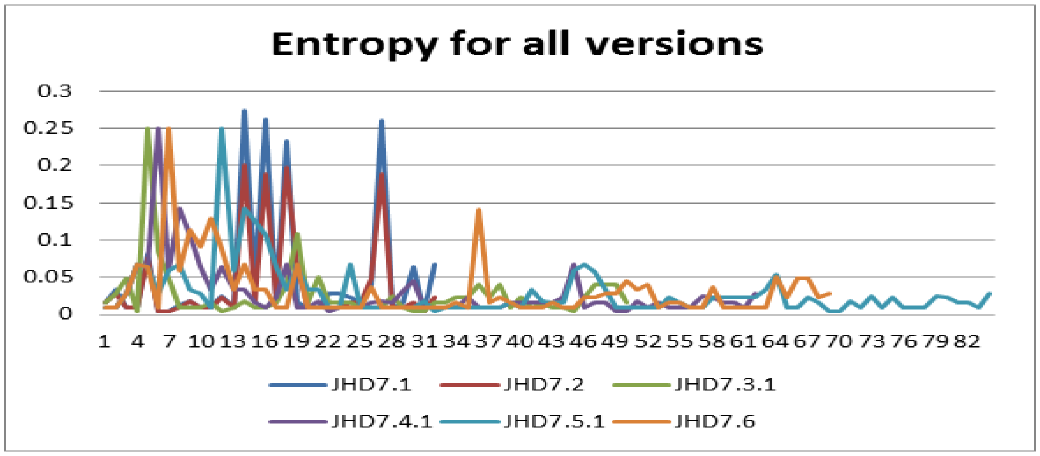

Figure 5. Entropy Values for all 6 versions of JHD

5. Analysis of Results

On close observation, it will be noted that all versions started with high entropy values, but as soon as the software transitions from JHD7.31 to JHD7.4.1, the entropy starts to drop and then stays consistently at a lower level. If we observe figure 4 above, we can also see that JHD attained its maturity during the transition from JHD 7.3.1 to V.7.4.1. According to [6], a software product reaches its maturity when software maturity index approaches 1. From both figure 6 and 7, we can theorize that in well-designed software that is based on best practices such as JHotDraw, the maturity level is reached at the turning point at which observed entropy starts to decrease. To allow us to visualize the adjustments JHD went through as it transitions and matures. The chart shown in figure 6 are constructed from data extracted from table 1.

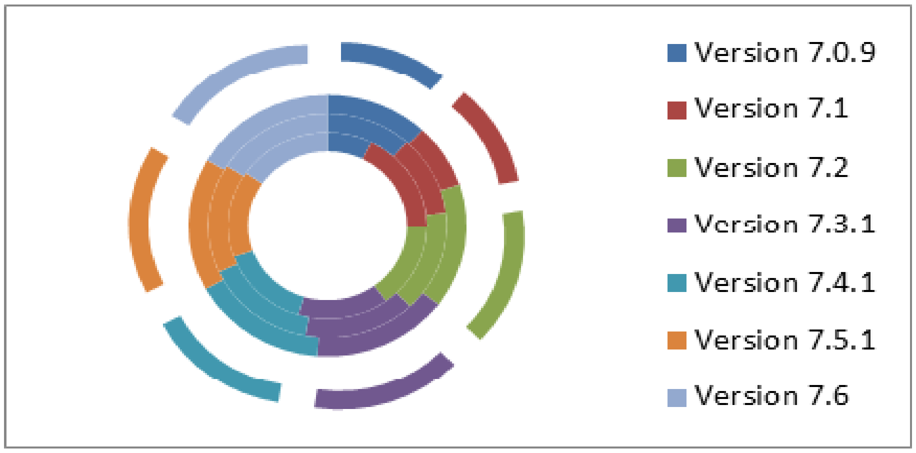

Figure 6. Blown-up Pie Chart for all the six Versions of JHD

It can be seen from the blown-up pie chart shown in figure 6 above that, when initially transitioning from (version 7.09 to 7.1 and from version 7.1 to 7.2), the pie parts in this transition did not align up properly with the outer pie pieces; however, as JHD evolves and transitions (clockwise), the pie pieces started to form perfect alliance with their respective outer pieces, indicating that maturity level has been attained, and the SMI remaining constant at 0.8 for (Version 7.4.1, Version 7.5.1 and the final Version 7.6).

When this observation is compared with the values of maturity index calculated from the static data collection, (see graph in figure 4 above), there is a correlation between the two results, in the sense that the expected maturity level is attained when JHD transitioned from (version 7.4.1 to 7.51); which is the point at which lowest entropy was reached and the highest software maturity index was attained. Another important observation is that, when JHD version transition static data (size, the number of classes, the number of methods and number of attributes) were graphed as shown in figure 7 below, it was observed that the number of methods consistently decreases as the software evolves and transitions from one version to another.

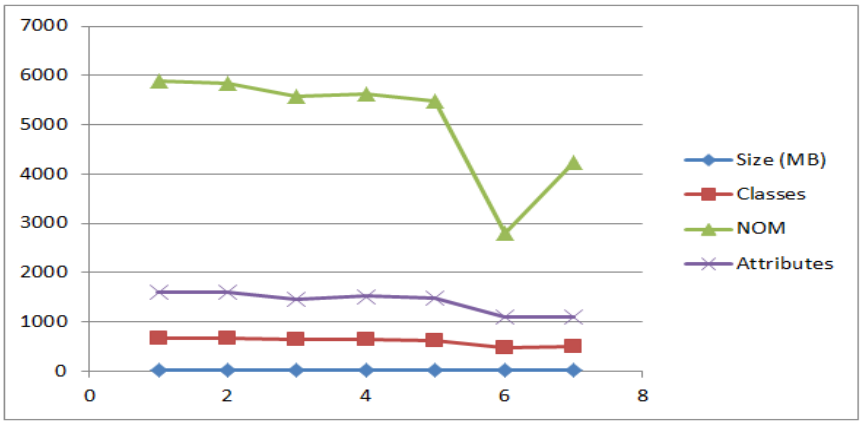

Figure 7. Correlation Between software size, number of classes, methods, and attributes

6. CONCLUSION

When a software system evolves and transitions from one version to another, it is expected that the new version will outperform the previous one and that the new version is better structurally containing fewer defects; however, this may not be the case, as new unintended consequences may be introduced, structure may be degraded and a measure of degradation and disorder may be introduced. This study is a first step towards investigating the behavior of a large-scale matured software system with a view to learn some lessons that can be used as a guideline in design, development, and management of new and existing software systems. In this work, it was consistently observed that JHD software components (classes, methods, and packages) that have undergone change or modifications in JHD evolution tend to generate higher entropy values than those with little or no unchanged; which is in line with an observation by [21] that, the most frequently invoked classes/methods in object-oriented software system are the ones that have the highest possibilities of being changed or modified. It is also observed that the entropy values consistently decreases as the software system evolves from one version to another, indicating that the software system was moving towards its optimal maturity level.

When JHD evolved few versions away from the last version, it is observed that the maximum maturity index attained was (0.8), confirming the statement made by [6] that, a software product reaches its optimal maturity level when its maturity index approaches the value of 1.0. In this research, when the optimal value of 0.8 SMI was reached, the entropy value remains stagnant with little or no change. Also, it was at this turning point that the JHD entropy level tends towards its lowest level, implying a possible correlation or connection between SMI and decrease in entropy, (i.e. decrease in degradation or disorder). In future efforts, we intend to study large-scale, middle-size and small-size object-oriented software systems that have gone through many versions with a view to finding some other hints that may generally be used as a maturity indicator, and a decision guideline for release readiness of software systems.

REFERENCES

[1] Martin and McClure, (1993). Software Maintenance: The Problems and Its Solutions Prentice Hall Professional Technical Reference 1983 ISBN:0138223610

[2] Alessandro Murgia1, A., Concas1, Pinna1, S., Tonelli1, R., Turnu, I. (2009). Empirical, Study of Software Quality Evolution in Open Source Projects Using Agile Practices

[3] Olague, H.M., Etzkorn, L.H., Cox, G.W. (2006). An Entropy-Based Approach to Assessing Object-Oriented Software Maintainability and Degradation-A Method and Case Study. ;In Software Engineering Research and Practice(2006)442-452

[4] Bianchi, A., Caivano, D., Lanubile, F., Visaggio, G. (2001). Evaluating Software Degradation through Entropy, Dipartimento di Informatica - Universith di Bari, Italy

[5] Jones, C. (2006). The economics of Software Maintenance in the Twenty-First Century Version 3 – February 14, 2006. http://www.compaid.com/caiinternet/ezine/capersjones-maintenance.pdf

[6] Pressman, R, Software Engineering - A Practitioner's Approach (6th Ed.). Newyork, NY: McGraw-Hill. p. 679.ISBN 0-07-285318-2

[7] Basili, R. and Rombach, H. D. (1988). The TAME project: Towards Improvement-Oriented Software Environments, IEEE Trans. on Software Engineering SE-14(6) (1988) pp.758–773.

[8] Becker-Kornstaedt, U., and Webby, R. (1999.) A Comprehensive Schema Integrating Software Process Modelling and Software Measurement, Fraunhofer IESE-Report No. 047.99 (Ed.: Fraunhofer IESE, 1999), http://www.iese.fhg.de/Publications/Iese_reports/.

[9] Bhattacharya, P., Iliofotou, M., Neamtiu, I., Faloutsos, M. (2012). Graph-Based Analysis and Prediction for Software Evolution Proceeding of the 34th International Conference on Software Engineering pp. 419-429

[10] Gao, Q., Li, J., Xiong, Y. et al. (2016). High-confidence software evolution. Sci. China Inf. Sci. (2016) 59: 071101. doi:10.1007/s11432-016-5572-2

[11] German, D. (2008). The Past, Present, and Future of Software Evolution. Frontiers of Software Maintenance, FOSM 2008.

[12] Bergmann R. and Eisenecker U. (1996). Case-based Reasoning for Supporting Reuse of Object-Oriented software: A case study, in Proc. Expert Systems 95 (1996) 152–169.

[13] Johari, K., and Kaur, A.(). Effect of Software Evolution on Software Metrics: An Open Source Case Study. ACM SIGSOFT Software Engineering Notes Page 1 September, Volume 36 Number 5, 2011.

[14] Deitel, H.M. & Deitel, P.J., (2003), Java How to Program, Prentice Hall, Upper Saddle River, NN, USA (1

[15] Giesecke (2006). Dagstuhl Seminar Proceedings 06301 Duplication, Redundancy, and Similarity in Software

[16] Mens, Kim., Kellens A., Tonella, P (2007), A Survey of Automated Code-level Aspect Mining Techniques, Transactions on Aspect-Oriented Software Development, Special issue on Software Evolution

[17] http://www.randelshofer.ch/oop/jhotdraw/index.html

[18] http://www.randelshofer.ch/oop/jhotdraw/Documentation/changes.html

[19] Hector M. Olague, Letha H. Etzkorn, Wei Li, Glenn W. Cox (2005). Evolution in software systems: foundations of the SPE classification scheme. Special Issue: IEEE International Conference on Software Maintenance (ICSM2005) Issue Overviews

[20] Opensource Software, www.sourceforge.net

[21] Joshi, P. & Joshi, R. (2006) Microscopic Coupling Metrics for Refactoring, IEEE Conference on Software Maintenance and Software Reengineering.

AUTHOR

Sayyed G. Maisikeli is currently an Assistant Professor at Al-Imam Muhammad ibn Saud Islamic University in Riyadh, Kingdom of Saudi Arabia. He obtained his dual Master of Science degrees in Computer Science and Operations Research from Bowling Green State University Ohio, and his Ph.D. from Nova Southeastern University in Florida, USA. His research interest includes Software evolution, Software visualization, Aspect mining, Software re-engineering and refactoring and Web Analytics.