# EVALUATION OF SOFTWARE DEGRADATION AND FORECASTING FUTURE DEVELOPMENT NEEDS IN SOFTWARE EVOLUTION

Sayyed Garba Maisikeli

College of Computer and Information Sciences
Al-Imam Muhammad Ibn Saud Islamic University
Riyadh, Kingdom Of Saudi Arabia

### *ABSTRACT*

*This article is an extended version of a previously published conference paper. In this research, 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 of this software were profiled, and data collected dynamically, from which four metrics namely (1) entropy (2) software maturity index, COCOMO effort and duration metrics were used to analyze software degradation, maturity level and use the obtained results as input to time series analysis in order to predict effort and duration period that may be needed for the development of future versions. The novel idea is that, historical evolution data is used to project, predict and forecast resource requirements for future developments. The technique presented in this paper will empower software development decision makers with a viable tool for planning and decision making.*

### *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, 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. When software evolves a lot can be learned such as complexity, degradation; this provides an opportunity to collect data that can be analyzed to project or forecast development duration and number of people (person-hours) required for future build.

Due to the need to deliver software products on time and the need to satisfy customer's satisfaction, software companies are compelled to release software at the optimal time. Gauging when software is ready to be released has been a very difficult factor to determine. At some point, a decision will be made that testing should be concluded and the product be released for customers use. The release decision is usually based on an evaluation of the software's expected quality balanced against its release date commitment [1].

This article is an extended version of a paper previously published by [2], in ICCSEA conference). Added to the previous paper are issues related to release readiness. In addition to entropy and maturity index, two more additional metrics (the COCOMO effort and duration) were added in the mix. The two more added metrics used COCOMO prediction model as presented by [3]. These additional metrics help us study and determine the effort required for each version build and consequently the expected time to build each version as the software evolves.

From JHotDraw data collected from the [4] archives, we already have the 'exact time duration' it took to build each version therefore, we can determine the optimal time from these two sets of data. The obtained results can then be used to forecast future build effort (person-hours) and the (time to build). Details of how these metrics were used are presented in the methodology section. According to [5], 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 [6], 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.

On the issue of measuring software degradation, [7 and 8] 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.

The rest of this article is organized as follows: Section 2 presents relevance of the research, section 3 discusses related research, section 4 presents the methodology used, section 5 presents analysis of results and section 6 concludes.

## 2. RELEVANCE

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. Other reasons that support the need for the study of software evolution include the consequences of ripple-effects, and providing guidance for the implementation of the software system. During transition of the software evolution, a lot of information can be deduced from the data collected; such as complexity, extendibility and degradation. With the incorporation of COCOMO effort and duration metrics and time series, software release readiness can be predicted, and the required resources such as (person-hours) and (development duration) can easily be projected and predicted. The predicted values equip and empower software development decision makers with a viable decision tools.

According to [9], 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 [9] 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 [10], 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. With regards to release readiness, [11] opined that, a poor understanding of the confidence in the quality level increases decision risk leading potentially to a bad release decision that possibly could have been avoided had the confidence in the quality been better known. A well-known critical system at jpl was used as a case study to investigate the value of certification to improve the mandated software readiness certification record (srcr) process.

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 and prediction of required resources for future evolutional developments cannot be taken lightly. If data collected during inter-evolution transitions is properly analyzed, valuable information can be deduced to forecast required resources for future evolution and implementation of the software system. This is what this paper sets out to achieve.

## 3. RELATED STUDIES

In a software evolution research, [12], 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, [12] 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, [9] 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 [13], 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, [14] 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, [15] 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.

[16] Presented an indicator which is sufficient for a mature software development organization to predict the time in weeks to release the product. [17] introduced the release readiness assessment where proprietary software is assessed on its ability to be released as open source/ open ecosystem.

In a statement, [18] believed that "software readiness is often assessed more subjectively and qualitatively, and stated that quite often, there is no explicit linkage to original performance and reliability requirements for the software, and that the criteria are primarily process-oriented (versus product oriented) and/or subjective. Such an approach to deciding software readiness increases the risk of poor field performance and unhappy customers". The author also stated that "unfortunately, creating meaningful and useful quantitative in-process metrics for software development has been notoriously difficult".

In a research work, [19] investigated the use of product measures during the intra-release cycles of an application. The measures include those derived from the Chidamber and Kemerer metric suite and some coupling measures of their own. the research uses successive monthly snapshots during systems re-structuring, maintenance and testing cycles over a two year period on a commercial application written in C++, and examined the prevailing trends which the measures reveal at both component class and application level.

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. In addition, the information collected is used to predict or forecast required resources for future evolution cycles as the software evolves.

## 4. METHODOLOGY

In addition to exploring and investigating 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 added two more metrics and incorporate time series analysis with a view to introducing a method of assessing software release readiness of various versions of a software system as it evolves from one version to another.

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. Six versions of our test software JHotDraw (JHD) were studied and analyzed in this research project.

### 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 [12] and [13] where they recommended the use of JHotDraw as an Aspect Mining validation benchmark. Also, [20] and [21] 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

Table 1. 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. [22], 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 [23], 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. [24] 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 [22]. 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.

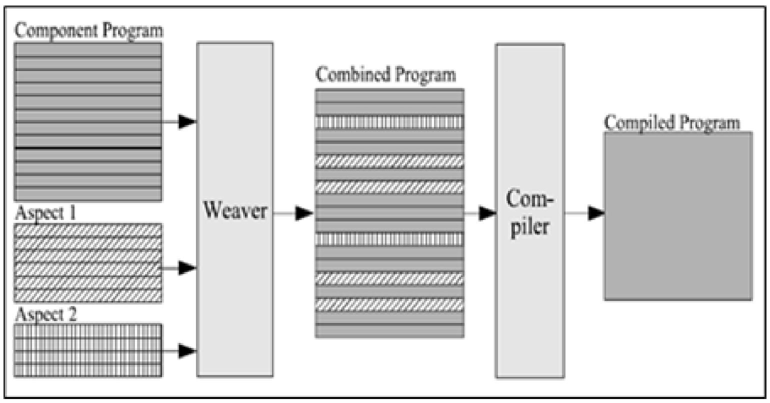


Figure 1. Example of how AspectJ Weaver works

### 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 [8], when investigating and studying the effect of a change in a software system, Shannon's equation may be better than complexity averaging. According to [5], in addition to measuring disorder introduced into software evolution, entropy also provides a measure of the complexity of the software system. [7], [27] stated that entropy can anecdotally

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 [8], when investigating and studying the effect of a change in a software system, Shannon's equation may be better than complexity averaging. According to [5], in addition to measuring disorder introduced into software evolution, entropy also provides a measure of the complexity of the software system. [7], [27] 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, [8] 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 expressed as follows:

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

Where

H = System Complexity Entropy,
pi = Probability that method mi 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 C1, C2, C3. Methods (m11, m21) are contained in C1, methods (m12, m22, m32) contained in C2, and (m13, m23, m33, m43, m53) contained in C3. The numbers shown beside class methods are representations of the frequency of method invocations when the test software was exercised.

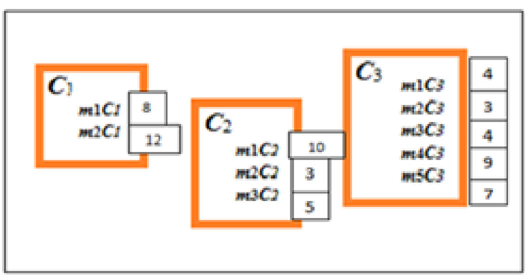


Figure 2. Example of method invocation from three different classes in (software S)

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 2. 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.

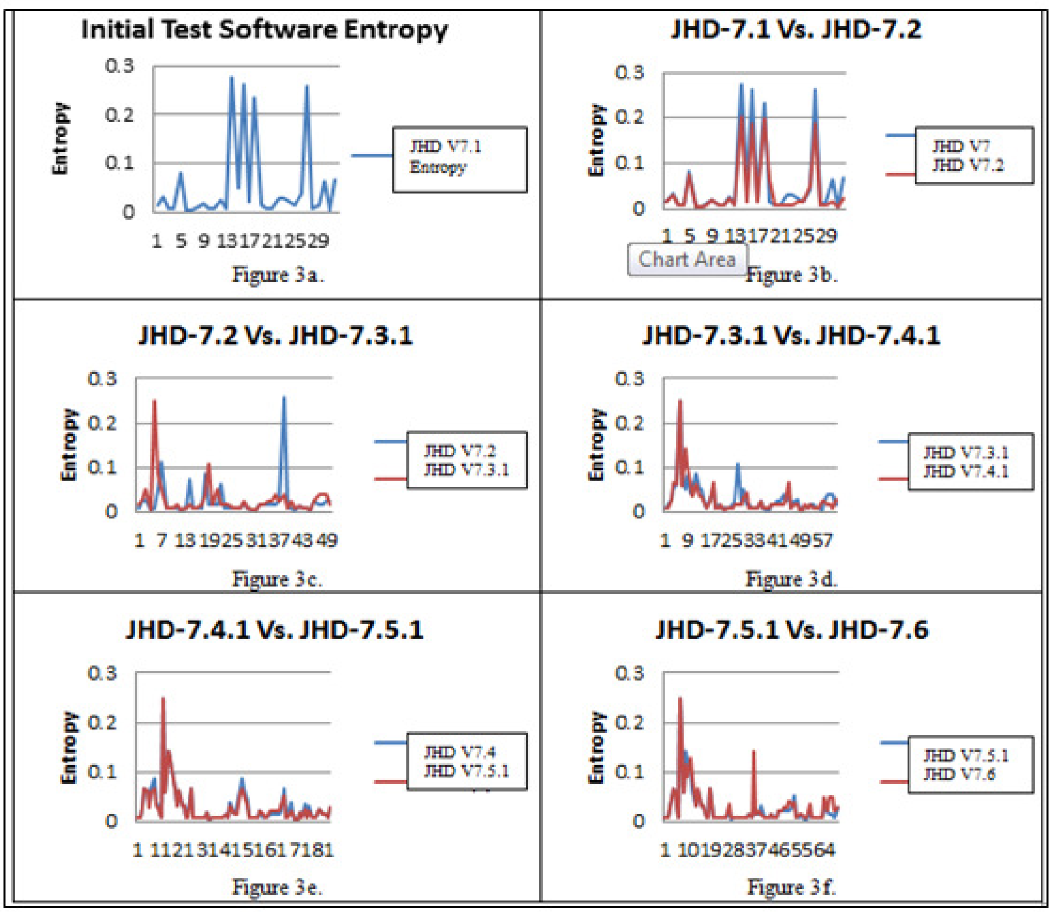


Figure 3 Entropy graph Version to Version

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, [10] 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 [10] 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 [9]. 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 [24] and [25], 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. 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).

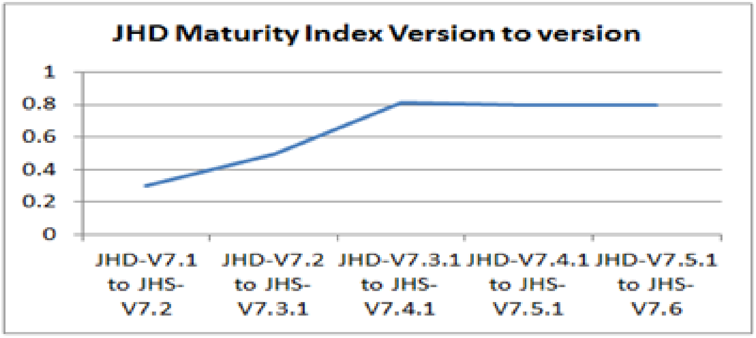


Figure 4 Inter-version Maturity Index

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.

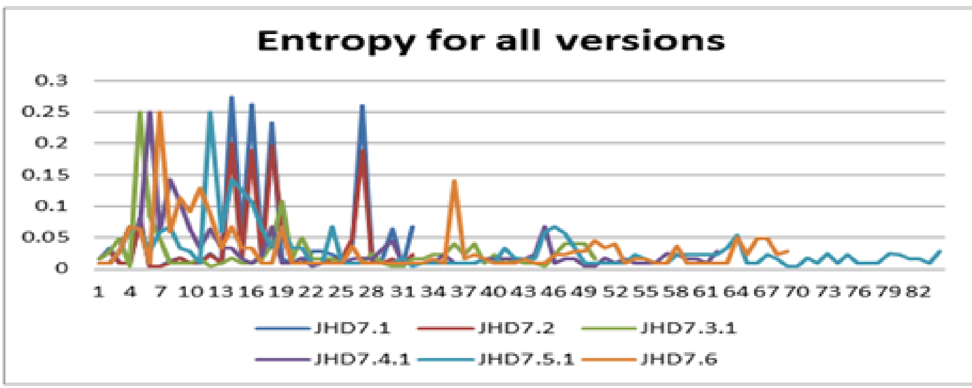


Figure 5. Entropy Values for all 6 versions of JHD

### 4.3.2 COCOMO Effort and Duration metrics

As mentioned in the introduction section, this paper is an extended version of the paper previously presented by [2]. In this paper, two COCOMO model metrics (effort and Duration) as presented by [3] were added to the metrics used in the previous paper. We used these additional metrics to help us determine the effort required for each version build, and the corresponding build time (period) for each version. It should be noted that data used for this purpose is archived at [25].

The COCOMO model for effort calculation uses the following formula.

$$E_n = a_1 * [Size]^{b1} \quad (1)$$

Where $E_n$ = Effort (in person-hours)

a =coefficient extracted from table 4 (based on the software category)

s= Size of software version (in LOC)

Table 4 Effort coefficients

| Software Category | $a_1$ | $b_1$ |
|---|---|---|
| Organic | 2.4 | 1.05 |
| Semi-detached | 3.0 | 1.12 |
| Embedded | 3.6 | 1.2 |

To calculate the Effort we used formula 1 above, and the associated coefficients were extracted from table 4 above. The Size (LOC) for different versions are shown in table 5 below. With these, the effort values for all versions are calculated. The derived effort values (person-hours) for each version are shown in column 3 of table 5. These values are then used as input to formula 2 below. This formula is used to calculate the duration for different version builds. Note that, since the time series dataset we are dealing with is not large, there is no need to remove the trend effect and seasonality of the data.

Table 5 Effort results

| Version | LOC | Effort Applied in Person-months |
|---|---|---|
| V1 | 52913 | 218751.86 |
| V2 | 53753 | 2822.96 |
| V3 | 71675 | 70188.67 |
| V4 | 73361 | 5866.94 |
| V5 | 72933 | 1390.68 |
| V6 | 79275 | 23580.27 |
| V7 | 80169 | 3013.81 |

### 4.3.3 Duration for building each version

To calculate the required time (duration) for building each version, we used person-hours (column 3 of table 6) above and substituted these values in formula 2 below. This produces time (duration) required to build each version as the software evolves.

$$D_n = a_2 * (E_n)^{b2} \qquad (2)$$

Where $D_n$ = Time (duration) required to produce version ($V_n$)
$E_n$ =Effort (persons-hours) required to produce version ($V_n$)

Note that the coefficients ($a_2$ and $b_2$) are extracted from table 6 based on the nature of the category of the software being tested. Since our test software is organic, the highlighted row coefficients in table 6 are selected.

Table 6 Coefficient selection

| Basic COCOMO model | $a_2$ | $b_2$ |
|---|---|---|
| **Organic** | 2.4 | 1.05 |
| Semi-detached | 3.0 | 1.12 |
| Embedded | 3.6 | 1.2 |

Table 6 The data presented in table 8 below was calculated from archived data and documentation for our test software, the (JHotDraw). Column 2 of Table 7 below represents duration for each of the 7 versions of our test software.

| Version | Duration | Persons-months |
|---|---|---|
| V1 | 267.376 | 818.14 |
| V2 | 51.1927 | 55.14 |
| V3 | 173.589 | 404.34 |
| V4 | 67.5984 | 86.79 |
| V5 | 39.1171 | 35.55 |
| V6 | 114.686 | 205.61 |
| V7 | 52.4812 | 57.43 |

With calculated effort $E_n$ (in person-months), and corresponding project duration for each version $D_n$ (Development time in months), it is then possible to calculate the number of people required for building a particular version. The required formula is as follows:

$$\boldsymbol{N}\text{n} = \frac{\boldsymbol{E}\text{n}}{\boldsymbol{D}\text{n}} \qquad (3)$$

Where $N_n$ =(number of persons)
$D_n$ = (person-months)
$n$ = 1..7 (in our case the seven versions of JHotDraw)

Column 3 of table 8 is the obtained results for number of persons required for building each of the seven versions. With the derived historical data, we are now ready to apply time series analysis to predict future, which is presented in the next section.

The obtained results Effort (person-hours) and Duration (build time), are then submitted to ARIMA time series analysis to predict or forecast the future build needs, effort (person-hours) and the duration (time to build) as the software evolves.

### 4.3.4 Time Series Analysis

A time series model is to obtain an understanding of the underlying forces and structure that is contained in the data, and is used to fit a model that will predict future behavior. In this model, data from past experience is used to forecast future events. Time series analysis predicts a response variable for a specified period of time. The forecast results are based on inherent or latent patterns that in exist in the data.

This paper utilizes the data collected during the transitional evolution transitional periods of our selected test software, the (JHotDraw), and subjected them to time series analysis with a view to being able to predict future required resources for future evolutionary development. Since we have historical data that spans (5 years), we have enough data to study trend patterns as well as being able to predict number of resources (people) and development period for future software evolutionary developments. If accurate and optimal effort (person-hours) and duration can be predicted then project budgeting and time to complete the future evolution projects can be determined, leading to the decisions about release readiness of a software system as the software system evolves from one version to another.

To have proper and accurate calculation of COCOMO model (effort and Duration), we thought of adjusting the LOC figure such that only lines of code added or deleted during the evolution are considered; however we realized that due to the principle of connascence, a change (additions, deletions or modification) in one part of a software may affect other parts, so the calculated

(COCOMO effort and duration) values were submitted to time series analysis as is, without any data adjustments.

Auto-Regressive Integrated Moving Average (ARIMA) time series and forecasting analysis tool was used to forecast future resources (person-hour) and (duration) that may be required for future evolutionary developments. The obtained results are shown in figure 6 below. The time series data is represented by blue line and the red line represents the predicted (duration) values.

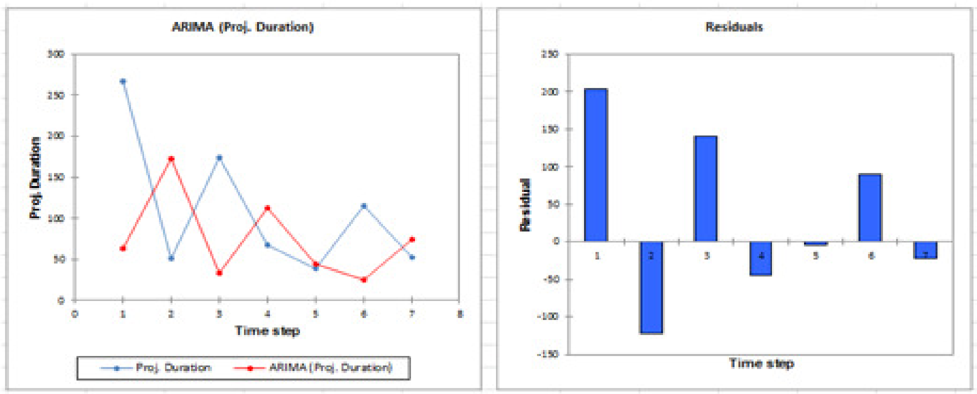


Figure 6 AMRIMA Extrapolation Forecast (future development duration)

Similarly, the historical effort (person-hours) calculated from (calculated from formula 3 above) was submitted to ARIMA analysis, and the required person-hours required for future development are forecasted and presented in figure 7 below.

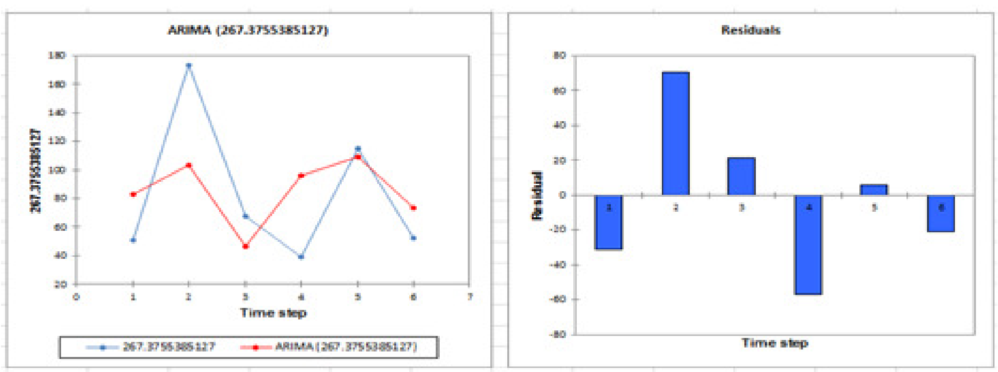


Figure 7 AMRIMA Extrapolation Forecast of future development Effort (person-hours)

With the two predicted values, project managers, planning managers, analysts and other decision makers can determine number of people required, and the duration for future development projects as the software system evolves.

## 5. Analysis Of Result

From the obtained data graphed in figure 4, it can be seen that maturity level for JHD is attained at the point at which lowest entropy was reached (transition from JHD 7.3.1 to V.7.4.1). Another important observation is that, when JHD version transition static data (size, the number of classes, the number of class methods and number of attributes) were graphed as shown in figure 8 below, it was observed that the number of class methods or (functions) in the test software consistently decreases as the software evolves and transitions from one version to another.

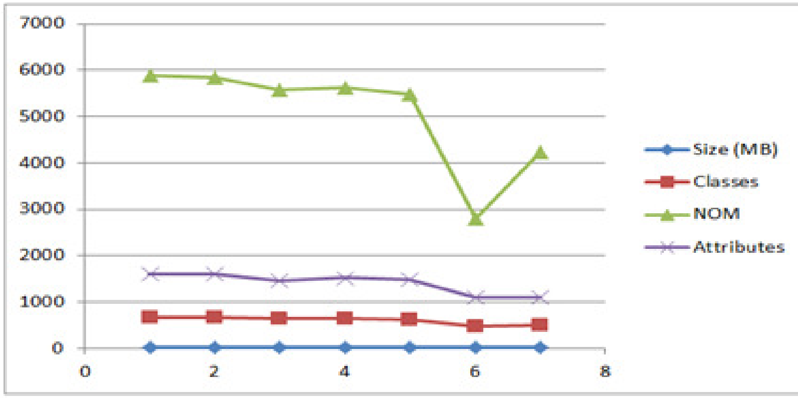


Figure 8. Correlation Between software size, number of classes, methods, and attributes

Data extracted from ARIMA analysis was used to construct table 10, this table shows resource predictions going forward. Columns 2 and 3 of table 10 can be used as a guide for planning future builds

Table 10 (Resource predictions)

| Future Transition | Development period (months) | People needed |
|---|---|---|
| 8 | 64 | 83 |
| 9 | 173 | 103 |
| 10 | 33 | 46 |
| 11 | 112 | 96 |
| 12 | 44 | 109 |
| 13 | 25 | 74 |
| 14 | 74 | |

## 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 investigated 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 during evolution tend to generate higher entropy values than those with little or no change; which is in line with an observation by [28] 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 [10] 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).

Although quality, reliability and availability issues are not addressed in this research, available data collected as the software evolves is used to study degradation, attainment of maturity level, possible connection between SMI were observed and addressed. With the introduction of time series analysis into the mix, this research presents a method that uses knowledge gained from past experience to forecast or predict resources such as (development duration, and number of persons) required for subsequent evolution of the software system.

In future, 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 guideline for determining release readiness of software systems, and monitoring software degradation as the software system evolves.

## REFERENCES


[1] Satapathy, P. R. (2008). Evaluation of Software Release Readiness Metrics Across the Software Development Cycle

[2] Maisikeli, S.G (2016). Evaluation and Study of Software Degradation in the Evolution of Six Versions of Stable and Matured Open Source Software Framework. Sixth International Conference on Computer Science, Engineering & Applications (ICCSEA 2016), Dubai, UAE, September 24-25, pp.1-13, ISBN: 978-1-921987-56-4

[3] Clark,B.COCOMOEffortModel, http://www.psmsc.com/UG1998/Presentations/cocomo2%201998.pdf

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

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

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

[7] 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

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

[9] 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

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

[11] Port, D., Wilf, J. (2011). The Value of Certifying Software Release Readiness: an Exploratory Study of Certification for a Critical System at JPL. System Sciences (HICSS) 2011 44th Hawaii International Conference on Vol. no., pp. 110, 2011

[12] 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.

[13] 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/.

[14] 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

[15] 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

[16] Staron, M., Meding, W., Palm, C. Agile Processes in Software Engineering and Extreme Programming Volume 111 of the series Lecture Notes in Business Information Processing pp. 93-107

[17] Kilamo, T., Hammouda, I., Mikkonen, T., Aaltonen, T., 2012. From proprietary to open source-growing an open source ecosystem. Journal of Systems and Software 85, 1467–1478.

[18] Asthana, A. (2009). Quantifying software reliability and readiness. IEEE International Workshop Technical Committee on Communications Quality and Reliability, 2009. CQR 2009.

[19] Ware, M. P., Wilkie, F. G. (2008). The use of Intra-release Product Measures in Predicting Release Readiness. First International Conference on Software Testing, Verification, and Validation, 2008

[20] Johari, K., and Kaur, A. (2011). 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.

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

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

[23] 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

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

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

[26] 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

[27] Opensource Software, www.sourceforge.net

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


**Authors**

**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