Does Software Design Complexity have Effects on Defects?

Normi Sham Awang Abu Bakar

Department of Computer Science International Islamic University Malaysia, Jalan Gombak, 53100
Kuala Lumpur, Malaysia

Abstract: Empirical analysis is an important area in software engineering. One of the ways to ensure the quality of any software is to measure the software using statistical analysis of various metrics from empirical data. The emergence of various open source software repositories has contributed tremendously to the advancement in the empirical software engineering research. This paper presents an empirical analysis using a set of metrics such as Structural, Data, Procedure and System Complexity, as well as Class Size on open source data. The results show that only CBO is significant in predicting defects.

Key words Empirical study, Object-oriented metrics, Defect analysis, Open source, Repository mining

INTRODUCTION

Empirical analysis of software engineering is an important research method that can add new knowledge to areas related to process and product improvement. Since it is based on observation, and reflects our actual experience with methods, tools and techniques, empirical research is closer to the real world than analytical or theoretical research (R. Harrison et al., 1999). Empirical software engineering research offers the opportunity to build and verify theories for software engineering (Lehman and Belady, 1976). Therefore, it provides steps towards a better understanding of our discipline.

An empirical approach to assessing software engineering technology, including industrial collaboration, began on large scale in the 1970s with the work of Victor Basili and his group at the University of Maryland (V. R. Basili et al., 1986; B. Boehm et al., 2005) Since then, there has been an increased focus on the importance of and approaches to applying empirical methods in software engineering research (V. R. Basili et al., 2007; D. E. Perry et al., 2000; D. H. Rombach et al., 1993).

Empirical Science concerns the acquisition of knowledge by empirical methods. However, what constitutes knowledge and the methods for acquiring it, rests on basic assumptions regarding ontology (what we believe to exist) and epistemology (how beliefs are acquired and what justifies them) (D. I. K. Sjøberg et al. 2007). Empirical research seeks to explore, describe and explain natural, social or cognitive phenomena by using evidence based on observation or experience. It involves obtaining and interpreting evidence by experimentation, systematic observation, interviews or surveys, or by careful examination of documents or artifacts.

An empirical body of evidence in empirical software engineering can be described as a set of studies, each performed under certain explicit conditions, for which both quantitative and qualitative, subjective and objective data have been collected and based on which certain conclusions and interpretations have been provided (L. C. Briand, 2007).

Generally, research conducted on commercial product development is obstructed by restricted access to the development process and selectively released data. Companies that commercialize their software products are in most cases not interested in sharing the product’s source code due to the risk of code spilling over to competitors or “software pirates”. In contrast, open source software projects show considerably high transparency of data for research due to their development practices. Researchers can easily obtain the system’s source code from repositories that host the project, for instance, SourceForge (www.sourceforge.net), GoogleCode (code.google.com) and GitHub (github.com), as well as from the websites of the open source projects themselves, such as Apache (www.apache.org), Mozilla (www.mozilla.org) and OpenBSD (www.openbsd.org)

Software metrics refers to a broad range of measurements for computer software. Although the terms “measure”, “measurement” and “metrics” are often used interchangeably, it is important to note the subtle differences between them. Within the software engineering context, a measure provides a quantitative indication of the extent, amount, dimensions, capacity, or size of some attribute of a product or process. Measurement is the act of determining a measure.

The IEEE Standard Glossary of Software Engineering Terms (IEEEStandard, 1990) defines metric as “a quantitative measure of the degree to which a system, component, or process possesses a given attribute.” In addition, software metrics can be divided into two categories: Structured metrics and object-oriented (OO) metrics.

Corresponding Author: Normi Sham Awang Abu Bakar, Department of Computer Science, International Islamic University Malaysia, Jalan Gombak, 53100 Kuala Lumpur, Malaysia.
E-mail: nsham@iium.edu.my
Research Questions:
The questions that need to be investigated in this research are:

1. Is there a general correlation between system (data + structural) design complexity and various types of
defects, and thus, can the level of defects be predicted early enough to undertake strategies to minimize them in
the final product?

2. Between structural, procedural and data complexity, which has the most influence and/or is the most
appropriate for predicting system defects?

Research Hypotheses:
Based on the research questions, the research hypotheses are formulated for the purpose of answering the
questions:

1. H1 : Increasing values of the Average System Complexity (Average Structural Complexity plus Average
Data Complexity) correlate with increasing post-delivery defect density.

2. H2 : Increasing values of the Average Procedural Complexity correlate with increasing post-delivery
defect density.

3. H3 : Increasing values of the Class Size correlate with increasing post-delivery defect density.

Background and Related Research:
One of the most important objectives of software engineering is to improve the quality of software
products. The quality of software can be defined in different ways but one of the most common definitions is the
number of defects that arise in the final product (N. E. Fenton and S. L. Pfleeger,1997), be it functional defects
or programming defects, that can cause problems to users. The aim of this research is to identify whether there
is any relationship between the number of discovered post-delivery defects and the design complexity of a
system, and whether measured design complexity may provide an estimation mechanism for post-delivery
defect discovery.

This paper focuses on the measurement of several metrics in software products, which are open-source
projects and are available freely from SourceForge.net. The projects chosen to be investigated were written in
Java. This research is designed to investigate selected problems in a snapshot version of open-source systems
rather than studying the evolution of systems.

In empirical software engineering, measurement theory plays a considerably important role in
understanding the science of software development. During the past decade, measurement theory has been
proposed and extensively discussed (N. E. Fenton and S. L. Pfleeger,1997; Zuse, H., 1991) as a means to
evaluate the software engineering measures that have been proposed in the literature, and to establish criteria for
the statistical techniques to be used in data analysis. Measurement theory is a considerably convenient
theoretical framework to explicitly define the underlying theories upon which software engineering measures
are based. This means that measures are not defined out of context and that the theories on which they are based
can be discussed, adapted and refined (L. C. Briand, 2007).

Much of the work being done in this topic is related to the application of measurement theory in empirical
software engineering. As such, Briand et al. attempted to measure the relationships between design measures
and software quality in object-oriented systems using Univariate Analysis and Multivariate Logistic Regression
Model. Their results show that many of the measures capture similar dimensions in the dataset, thus reflecting
the fact that many of them are based on similar principles and hypotheses (L. C. Briand, 2007). In their work,
Basili et al. discussed the application of measurement and metrics as quality indicator in object-oriented systems
(V. R. Basili et al. 1996). In addition, there are many examples of the application of measurement theory in
empirical software engineering, to name a few, (Arisholm, E. et al, 2004; Basili, V. R. et al,2002; El-Emam, K.,
and A. D. Carleton, 2004) and many more.

The McCabe’s Cyclomatic Complexity metric (McCabe, T. J., 1976) was designed to indicate a program’s
testability and understandability (maintainability). It is the classical graph theory cyclomatic number, indicating
number of regions in a graph. As applied to software, it is the number of linearly independent paths that
comprise the program and can be used to indicate the effort required to test a program. To determine the paths,
the program/module procedure is represented as a strongly connected graph with unique entry and exit points.
For a program with flowgraph G, the general formula to compute cyclomatic complexity is:

\[ V(G) = e - n + 2p \] (1)

where
\[ V(G) \] = Cyclomatic number of flowgraph G
\[ e \] = Number of edges
\[ n \] = Number of nodes
\[ p \] = Number of unconnected parts of the graph
Another important metric to be considered in this research is one of the metrics introduced by Chidamber and Kemerer (S. R. Chidamber, and C. F. Kemerer, 1994), which is known as Coupling between Object Classes (CBO). The Chidamber and Kemerer object-oriented (CK OO) metrics suite consists of six metrics which use the object-oriented concept for the calculation, and they are: Weighted methods per class (WMC), Depth of inheritance tree (DIT), Number of children (NOC), Coupling between object classes (CBO), Response for class (RFC), Lack of cohesion metric (LCOM), however, only CBO is relevant in this paper.

**Card and Glass Model:**
This research is based on the earlier work of (Card and Glass, 1990) who argue that design measures are indicators/estimators of decision counts (cyclomatic complexity), module size (executable lines of code), and errors (discovered from system tests) (Card and Glass, 1990). In a similar line, this thesis investigates how metrics available during the detailed design phase (Data Complexity) can be used to estimate the number of decisions (Procedural Complexity) during the implementation phase, and Class Size during the coding phase.

Card and Glass hypothesized that the complexity of a system can be broken down into 3 main components: data, structural and procedural complexity (all established as part of design). According to Card and Glass, system design prescribes the strategy for implementing the requirements. The difficulty of that strategy results in the possibility of the developers making errors. Hence, system design tends to be an early source of errors (Card and Glass, 1990). Due to the fact that coding is largely a translation process once design is complete, most software complexity resides in the design. They carried out a study of 8 similar projects taken from the Software Engineering Laboratory database and all projects were “ground-based attitude determination systems for spacecraft in near-earth orbit”, written in FORTRAN (RATFOR). Some projects did not provide a complete set of design materials, therefore, many parts of the study relied on design product data extracted from software source code.

Based on various approaches to structure complexity and module complexity measures, (Card and Glass, 1990), developed a system complexity model.

\[ C_t = S_t + D_t \]  
where
- \( C_t \) = Total System complexity
- \( S_t \) = Total Structural complexity
- \( D_t \) = Total Data complexity

They defined average system complexity as

\[ S_{sys} = C_t/n = S_t/n + D_t/n \]  
where
- \( S_{sys} \) = Average system complexity
- \( C_t \) = Total system complexity
- \( S_t \) = Total structural complexity
- \( D_t \) = Total data complexity
- \( n \) = Number of modules in system

Structural complexity is further defined as

\[ SC = \frac{\sum f(i)}{n} \]  
where
- \( SC \) = Average Structural complexity
- \( f(i) \) = Fan-out of module i
- \( n \) = Number of modules in system

They further defined data complexity as

\[ DC = \frac{V(i)}{f(i) + 1} \]  
where
According to Card and Glass, system complexity is a sum of structural complexity and overall data complexity. Structural complexity is defined as the mean (per module) of squared values of Fan-out. Fan-out is the number of local flows that emanate from the given module plus the number of data structures that are updated by the module, while Fan-in is the number of local flows that terminate at the given module plus the number of data structures from which information is retrieved by the module. According to the findings in the literature (Card and Glass, 1990), the systems they studied mostly have either low or no Fan-in value, therefore, Fan-in is not considered as an important complexity indicator. Data complexity of a module is defined as a function that is directly dependent on the number of I/O variables and inversely dependent on the number of Fan-outs in the module as shown in Equation 5.

The rationale is that the more I/O variables in a module, the more functionality needs to be accomplished by the module and therefore, the higher internal complexity. In contrast, more Fan-out means that functionality is deferred to modules at lower levels, therefore, the internal complexity of a module is reduced. Finally, the overall data complexity is defined as the average of data complexity of all new modules.

The Structural Complexity in Card and Glass’s model, as presented in Equation 4 only considers one dimension of coupling, which is Fan-out (based on calls/invocation). In object-oriented (OO) systems, coupling measures should also include “inheritance” which is an important characteristic of OO systems. Furthermore, the value of Fan-out squared may not explain the complexity in OO systems as it might do in structured systems. Therefore, it is essential to find a substitute measure for coupling in OO systems.

In this research, the Coupling between Object classes (CBO) measure proposed by (Chidamber and Kemerer, 1994) is chosen as a possible alternative to CG Structural Complexity in measuring the Structural Complexity of OO systems. The reason for choosing CBO to measure Structural Complexity is that it captures both kinds of dependencies by incorporating the “accesses”/“uses” from the given class (Fan-out) and other classes (Fan-in), including “inheritance”.

This suggests an alteration of Card and Glass’s formula for Structural Complexity to make it more relevant to OO systems, that being:

\[
SC = \frac{\text{CBO}(i)}{n}
\]

where

\[
S = \text{Average Structural Complexity}
\]

\[
\text{CBO}(i) = \text{CBO of class } i
\]

\[
n = \text{Number of classes in system}
\]

A designed class will eventually be implemented in code. It is extremely useful to be able to predict the size of the class based on early design metrics. The Class Size metric is obtained using this formula:

\[
CS = \frac{\text{eLOC}}{n}
\]

where

\[
CS = \text{Class Size}
\]

\[
\text{eLOC} = \text{Effective lines of code in system}
\]

\[
n = \text{Number of classes in system}
\]

Research Methodology:

Data speak a thousand words. The backbone of empirical theory is data, either qualitative or quantitative, and without data, measurement work cannot be carried out. In empirical software engineering, data play an important role in theory validation, hypothesis testing, model building, etc.

The main reason open source software is chosen as empirical data in this research is because it allows researchers to access project source code, artifacts and other details of the software through several online repositories which offer publicly available data source of a size, diversity and complexity not previously available. In the past, acquiring data for research purposes has been difficult since not all organizations are willing to share their confidential data for public access to protect the confidentiality of the data from competitors or were even afraid that the data will be misused by the people who acquire the data themselves. In
contrast, open source software is readily available online to be collected, analyzed, shared and used by any interested party (N. S. Awang Abu Bakar, 2009).

Out of the three major open source repositories, i.e., SourceForge, GoogleCode and GitHub, based on the range of selections and the information included, SourceForge was selected as the source of data in this paper.

The projects in SourceForge are grouped into several categories based on functionality, such as communications, database, education, games, internet, multimedia, office/business, security, software development and many more.

For the purpose of this research, 106 projects were selected based on several selection criteria. In order to automate the searching process, a Python script which was written to identify systems that fulfill the following selection criteria:

1. Active projects (Activity percentile more than 90 percent).
2. Developed using Java programming language.
3. Development status is: Production/Stable or Mature.
4. High number of total downloads (number of downloads more than 50,000).
5. Availability of error reports (error reports shown in bug tracking system).

In order to select the relevant systems to be included in this study, all of the aforementioned search criteria need to be fulfilled. Thus, there is a need to build a tool that can automate the selection and at the same time, saves a lot of time to complete the process. The framework of the open source repositories mining is illustrated in Figure 1.

Fig. 1: Repositories mining framework

The tool interface is depicted in Figure 2. The search keywords are used to assist users to determine the types of systems they need to look for. For example, the Category keyword will narrow down the search within specific categories in SourceForge, and will return the results in a list as shown in Figure 3.

The results in Figure 3 are useful for users who need to download numerous systems for their work. Users can choose from a list of systems that are relevant to their needs. This tool can automate the filtering of systems and therefore, saves time compared to having to do the filtering manually. A bug report column is also added to help users to identify the systems which publish the bug reports in their SourceForge page.

After the selected systems have been downloaded, a metric extraction tool known as JHawk was used to obtain the important metrics for the defect analysis. The metrics are further discussed in the next section.

**Dependent Variable:**

In spite of recent advances in programming technology, it is not yet possible for developers to produce error-free code consistently. A software product is considered defective when it does not perform its functions according to the user’s expectations.

In this research, defects refer to faults in the system, that later will cause failures in the system. The dependent variable used in this research is post-delivery defects collected from the bug tracking report in SourceForge.
Independent Variables:
The independent variables selected for this research are a collection of design metrics that can be used to measure system complexity at the design phase of the software development. They are based on the system complexity model by (Card and Glass, 1990).

Structural Complexity (SC) is represented by CBO, Data Complexity (DC) is represented by Number of Parameters and Procedural Complexity (PC) is represented by McCabe’s Cyclomatic Complexity, lastly, another metrics to be considered is Class Size (CS). Specifically, the metrics used in this work are measured at the system and class level.

Fig. 2: Search page

Fig. 3: Result list page

RESULTS AND DISCUSSION

The initial analysis on the data shows that the distribution is not normal. Therefore, for correlation analysis, Spearman correlation analysis was done. The overall results of the Spearman analysis are shown in Table 1.
In Table 1, the top row represents the values for the Spearman correlation coefficient between the two variables, while the bottom row (in parenthesis) represents the p-value for the correlation. The results in Table 1 shows that only Structural Complexity is significant in predicting defects (Spearman corr. = 0.29, p-value < 0.001). However, System Complexity is not significant in predicting defects and the relationship is very weak (Spearman corr = -0.155, p-value = 0.114). Based on this result, we can conclude that H1 is rejected, which means there is no correlation between System Complexity and Defects. Similarly, Procedural Complexity and Class Size also show no correlation with defects (Spearman corr = 0.045, p-value = 0.649) and (Spearman corr = 0.011, p-value = 0.909). This means that H2 and H3 are also rejected.

The correlation between defects and Structural Complexity is depicted in Figure 4. The fit line is a bit counter-intuitive because we would expect to see that if complexity rises, defects will also rise. However, in Figure 4, the result is quite the opposite. This could be attributed to the possibility that designers and developers take extra care on the complex modules, which could make them less error-prone. The rest of the metrics are shown in Figure 5 to 8. Figure 5 to 7 show that there is almost no relationship between the two variables.

![Fig. 4: Structural Complexity vs Defects](image-url)
Fig. 5: Data Complexity vs Defects

Fig. 6: Procedural/Cyclomatic Complexity vs Defects
**Fig. 7:** Class Size vs Defects

**Fig. 8:** System Complexity vs Defects

**Conclusion:**
This research explores the relationships between variables such as Structural Complexity, Data Complexity, Procedural Complexity and Class Size with defects. The results show that only CBO is correlated with defects and surprisingly, the correlation is not in the expected direction. System Complexity is not correlated with defects and suggested by Card and Glass.
This research can be further expanded to include C++ systems in addition to the Java systems being included in the current analysis. This should be able to help developers understand what they can control in order to minimise defects.

REFERENCES


code.google.com


github.com


www.apache.org

www.mozilla.org

www.openbsd.org

www.sourceforge.net.