

# Variation of Defects and Complexity in Multiple versions of Open-Source Projects

Kshitij Shukla, Clive Boughton (Supervisor)

Department of Computer Science

Australian National University

Canberra, Australia

[u4513469@anu.edu.au](mailto:u4513469@anu.edu.au), [clive.boughton@anu.edu.au](mailto:clive.boughton@anu.edu.au)

## Abstract

Studies has showed that the number of pre delivery defects increases with the complexity of the system. The aim of our research is to establish whether such a phenomenon applies to the post-delivery defects. We do this by studying and analysing different software products and the number of reported post-delivery defects for their respective versions. Our research is currently based on the work of Card and Glass [2], Awang Abu Bakar and Boughton [1] who argued that software system complexity measures are predictors of defects. The work of Card and Glass [2] is based on structural design software and the work of Awang Abu Bakar and Boughton [1] is with object-oriented software. We studied five open-source, object oriented systems downloaded from SourceForge.net [14]. Our initial result demonstrates that system complexity does predict post-delivery defects, but that the relationship is not as strong as for pre-delivery defects. We also demonstrate how various indicating parameters of a open source software changes with versions.

Keywords: system complexity, software metrics, open source software, CBO, Fanout, Fanin, post-delivery defects, parameters

## 1. Introduction

“Finding and fixing a software problem after delivery is often 100 times more expensive than finding and fixing it during the requirements and design phase.” –CeBASE, Dec, 2002.

Almost all companies nowadays are concerned about the quality of their software. However, there are different ways to define the quality of software. The one of the most common definition for quality of a software product is: the number of defects that arise in the final product [4], be it functional defects or programming defects that have caused the problems. Companies now-a-days release products after fixing the defects that they have found prior to release and then, with time, they release newer versions of the same software by fixing errors as pointed out by different users, and also by adding new functionalities. Few companies attempt to predict defects based on earlier development artefacts such as design models.

Data collection (in our case, all versions of software and post-delivery defects) is an important task, especially when it has to be collected from the vast collection of projects in SourceForge.net [14]. Thus, the data collection process has to be done using a systematic process to ensure that measures are defined unambiguously, that collection is consistent and complete.

Selecting a software metric tool to be used, and validating it, is also an important part of the process, since the whole project depends upon the result given by the tool selected. We have selected four tools from the many available on the market. All selected tools can be used to produce software metrics for systems written in Java, they are: JStyle [12], JHawk [11], Chidamber and Kermerer Java Metrics (CKJM) [10] and Resource Standard Metrics (RSM) [13]. After analysing the result from these metrics tools, we selected one that gives results

closest to our definition and understanding of ‘object oriented software metrics’.

There are many open source systems available in SourceForge.net [14]. We downloaded most of the versions of five selected Java-based software products for analysis, depending upon higher number of downloads, development status and activity percentile. Most of these software products are listed in most active project lists in SourceForge.net [14].

## 2. Background

Open source software is commonly accepted and successfully adopted by many organizations, and some of these systems have been used for mission critical purposes [4]. Also, a study by Zhou and Davis [9] shows that open source projects show a similar reliability growth pattern to that of proprietary software projects. Thus, even though open source development methodologies are different than proprietary software development methodologies, they have similar properties and thus they can be treated as an indicator of software quality.

According to Paulson et al. [8], creativity is more widespread in open-source projects, and defects are found and fixed more rapidly, compared to closed-source projects. This is one of the reasons for selecting open-source software for our research work. According to Lincke et al. [6] existing software tools interpret and implement the definition of object-oriented software metrics differently. That’s why we chose to validate the metrics tools, before actually using them for software analysis.

According to Card and Glass [2], system complexity (sum of data complexity and structural complexity) is an indicator/predictor of defects (found in system testing). In their work, they chose pre-delivery defects and structured system oriented software. All eight software systems were written in FORTRAN (RATFOR) in the Software Engineering Laboratory, and sponsored by NASA Goddard Space Flight Centre (GSFC). Awang Abu Bakar and Boughton [1] continued the work by Card and Glass, [2] by examining post-delivery defects for object oriented software system. They studied one

version of each of the 10 projects randomly selected.

Our work is a continuation of Awang Abu Bakar and Boughton’s [1] work, trying to prove in that design/system complexity is a predictor of defects (post-delivery). The main difference is that we studied five systems, taking multiple releases of the software into consideration.

## 3. Metrics Tool Validation

There are many tools available in the market and they nearly all give different results for complexity elements because they all employ different assumptions on the way to calculate various metric values [6]. We chose four tools and tried to match the results given by them, with our manually-calculated results. The main idea of this process was to identify which one of the given tools gives the result closest to the results calculated by us.

The tools which we used were:

1. Chidamber & Kermerer Java Metrics (CKJM) [10]
2. JStyle [12]
3. JHawk [11]
4. Resource Standard Metrics (RSM) [13]

These tools analyse and produce many results and parameters but the parameters that are useful to us were:

1. Structural Complexity (Fanout or CBO)  
According to Henry and Kafura [5], Fanout is the number of local flows out of a module plus the number of data structures that are used as output.  
According to Chidamber & Kermerer [3], the CBO of a class is a count of the number of other classes to which it is coupled.
2. Data Complexity (Number of (class) parameters)  
According to Card and Glass [2], data complexity of a module is the number of data items (variables) it is expected to process.
3. Procedural Complexity (Decision count or Cyclomatic Complexity)

According to McCabe [7], cyclomatic complexity is the measure of control flow within the module.

Out of all the tools, only JHawk produces all the required parameters. The others produce respectively:

1. JStyle only gives Fanout and Cyclomatic complexity.
2. RSM only gives numbers of parameters and Cyclomatic complexity.
3. CKJM only gives CBO.

For the validation purpose, we chose five open source Java software products downloaded from SourceForge.net [14].

### 3.1 Calculating Fanout

As only JHawk and JStyle produce the Fanout parameter, we compared the values given by both these tools against our manually-calculated values. For the validation purpose, we chose one version each of 5 Java open source software projects, i.e., DataCrow, Galleon, HTMLParser, Cewolf, Gantt-Project and YALE (downloaded from SourceForge.net [14]). Out of all these versions, we selected five class files (Files with .java extension) each for analysis.

According to our calculation (Manual) and our definition, fanout should include:

1. Method Calls for a different class.
2. Definition (New call) and Declaration of an object of different class.
3. Include variable calls from different class.
4. Interface classes (instance of and implements function call)
5. Throw statements for exception classes
6. Type conversion
7. If one or more of the above calls is done for the same class then fanout should only be increased by one.

And fanout should not include:

1. Any calls to \*.java and \*.javax classes
2. Any external classes which are imported but not in the source folder of the software, i.e.,

that are not coded by the developer but are imported in the class.

3. Inherited classes.

Figure 1 is a plot of the Fanout values calculated manually & the same value from JHawk, Figure 2 shows the smiler plot for JStyle, respectively. Comparing the results given by Figure 1 and Figure 2, we reached the conclusion that the correlation coefficient for JHawk and Manual calculation (0.821) is better than that of JStyle and Manual Calculation (0.698). That symbolises the fact that JHawk values are more closely related to our manual calculated values, as compared to JStyle values.

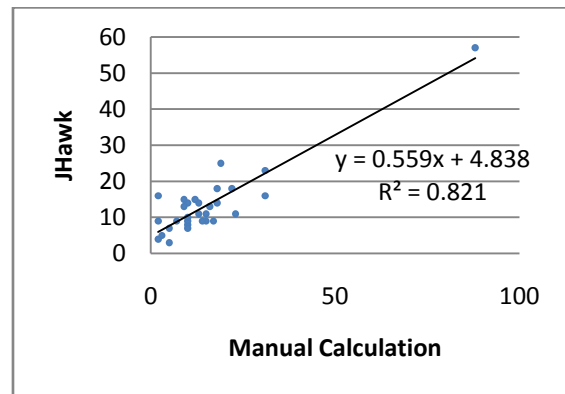


Figure 1: Fanout values, JHawk v/s Manual Calculation

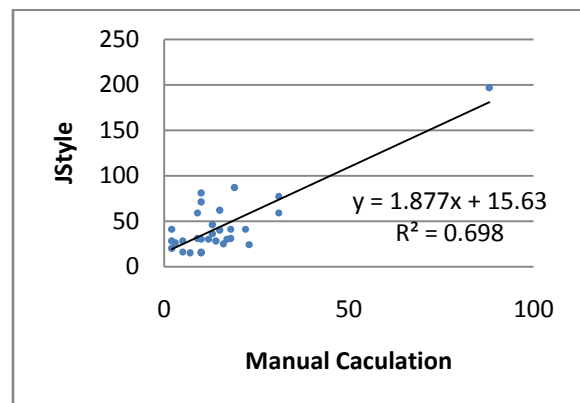


Figure 2: Fanout values, JStyle v/s Manual Calculation

We tried to analyse the tools more deeply, to find why the tools are giving different results compared to our manual calculation. We found the following:

JStyle includes:

1. Any calls to \*.java and \*.javax classes
2. Any external classes which are imported but not in the source folder of the software, i.e., that are not coded by the developer, but are imported in the class.
3. Inherited classes.
4. Unique method Calls for different methods of the same class.
5. Definition and Declaration of an object of different class.
6. All the variable calls from one class as one.
7. All these calls are considered separately even if they are for the same class and thus counted more than one time.

JHawk includes:

1. Method Calls for a different class.
2. Definition (New call) of an object of different class.
3. Include variable calls from different class.
4. Interface classes (instance of and implements function call)
5. Throw statements for exception classes
6. Any external classes which is imported but not in the source folder of the software i.e. which is not coded by the developer but is imported in the class.
7. If one or more of the above calls is done for the same class then fanout should only be increased by one.
8. If class is reference to itself then it will count itself also.

### 3.2 Calculating CBO

We followed the same process used in Calculating Fanout. This time we compared CBO value by JHawk and CKJM (only these two produces CBO parameter) with manually calculated values by us. For the validation purpose, we chose one version each of five java open source software projects, i.e. DataCrow, HTMLParser, Cewolf, JasperReport and FreeMind (downloaded from SourceForge.net [14]).

According to our calculation (Manual) and our definition, CBO should include:

1. Fanout definition as in the previous section.
2. Fanin, According to Henry and Kafura [5], Fanin is the number of local flows into a module plus the number of data structures that are used as input.

We calculated Fanin by writing a shell script that gives us output in the form of number and names of classes calling the class for which we were calculating the CBO.

3. Inheritance i.e. Extent call in Java
4. CBO is calculated from union of Fanin, Fanout and Inherited class sets, i.e. if the class is present in more than one set, then it is counted only once.

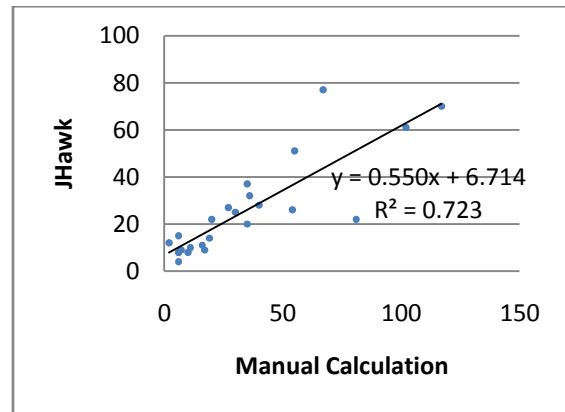


Figure 3: CBO value, JHawk v/s Manual Calculation

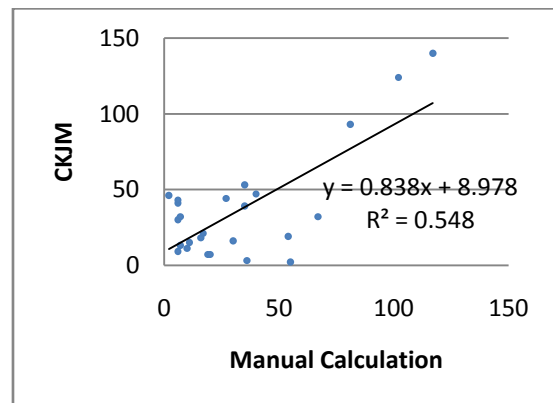


Figure 4: CBO values, CKJM v/s Manual Calculation

Figure 3 is a plot of the CBO values calculated manually & the same value for JHawk, Figure 4, shows the similar plot for CKJM. Comparing the result given by Figure 3 and Figure 4, we reached the conclusion that the correlation coefficient for

JHawk and Manual calculation (0.723) is better than that of JStyle and Manual Calculation (0.548). That symbolises the fact that JHawk values are more closely related to our manual calculated value as compared to JStyle values.

### 3.3 Calculating Cyclomatic complexity

As JStyle, JHawk and RSM produces cyclomatic complexity value for the software, we tried to compare these values to our manually calculated values. We followed McCabes [7] definition of calculating Cyclomatic complexity. According to our definition Cyclomatic complexity should be increased by one for:

1. Return call
2. Break call
3. Case function in switch
4. Sequence

And it should be increased by 2 for:

1. If Then Else statement
2. While loop
3. Until or For loop

After analysis of the result given by JStyle, JHawk and RSM, we came to the conclusion that JStyle produces the closest result as per our manual calculation, and thus we decided to use JStyle to calculate cyclomatic complexity for the five software systems that we chose for our research.

### 3.4 Calculating Number of Parameters

We followed the same process used in Calculating Fanout and CBO. This time we comparing value by JHawk and RSM (only these two produces number of parameters value) with manually calculated values by us. For the validation purpose, we chose one version each of five Java open source software projects i.e. DataCrow, Galleon, HTMLParser, Cewolf, GanttProject and YALE (downloaded from SourceForge.net [14]).

What we found was that the value produced by both of them is the same as the value calculated by the manual calculation.

### 3.5 Tool Selection

After the complete analysis and validation of the tools, we found that, overall JHawk gives Fanout, CBO and number of parameter values closer to our manual calculation. Thus, we chose JHawk to calculate CBO and number of parameters, and since JStyle gives value closer to our value for Cyclomatic complexity, we chose JStyle to calculate cyclomatic complexity.

## 4. Software System Data Analysis

The data analysis part is divided into two steps:

Step 1:

This step involved selecting five open source Java software from SourceForge.net [14], to be used for analysis. It also included feeding the source code of different software releases to the JHawk/JStyle (metrics-calculating tool) and collecting all the values of the necessary fields after these tools produced their metrics results, which were used later. Also, the post-delivery defects information was collected from the software product websites.

Step 2:

This step involved data analysis to find some trends and relationships for all the data collected. We also used statistical packages to help us to analyse the data more effectively.

We started Step 1 by selecting software to be used to analyse the trends. The main criteria of selecting a software product were:

1. It should be an active project i.e. there Activity Percentile > 90%.
2. It should have a high number of downloads i.e. Total Number of Downloads > 50,000.
3. Its Development status should be productive or stable.
4. It should have a significant number of errors reported with every version release.
5. It should be a Java Project.

After careful consideration we selected the following five software products:

1. *Data Crow*: The ultimate movie & video, book, images, games, software and music catalogue. Uses online services (Amazon, imdb and many others) and parses file information (mp3, divx, etc).
2. *HTMLUnit*: Is a "browser for Java programs". It models HTML documents and provides an API that allows you to invoke pages, fill out forms, click links, etc.
3. *Saxon XSLT and XQuery Processor*: The Saxon package is a collection of tools for processing XML documents
4. *LaTeXDraw*: Is a graphical open source PSTricks code generator or a PSTricks editor for LaTeX distributed under the GNU GPL.
5. *ZK- Simply Ajax and Mobile*: ZK is Ajax Java framework without JavaScript. With direct Rich internet applications (RIA), 200+ Ajax components and mark-up languages, developing Ajax/RIA as simple as desktop apps and HTML/XUL pages.

It was difficult to analyse each and every release of these software products, as many versions were minor releases and were done very frequently. So, we didn't expect too many changes in the software in context with its complexity and the number of defects reported. To solve this problem, and instead of analysing all the versions released, including major and minor, we selected versions with a gap of at least 3 months in their releasing dates.

We also collected the number of post-delivery defect reports from the 'Bugs' section of SourceForge.net [14]. While collecting the number of post-delivery defects reported, we only selected those defects that were being fixed or were in the process of being fixed, i.e., all the defects that were genuine and unique, and not just a problem that the user was having because he/she is new to the software. All information regarding the defects were collected and kept for later use.

The next step involved feeding all the selected versions of the software products into the metrics-calculating tool, and collecting the required field values. The variable we were interested in were

CBO, number of Parameters passes, Cyclomatic complexity and number of Classes.

According to Card and Glass [2], work performed (within modules) as well as the connections among the work parts (modules) are the constituents of system complexity.

It can be calculated by the formula:

$$Ct = St + Dt$$

Where:

Ct = System Complexity

St = Structural (inter-module) complexity

Dt = Data (intra-module) complexity

Thus the System complexity (Ct) can be defined as the sum of intra-module and inter-module complexity. As we are dealing with object oriented software, System complexity can be defined as sum of inter-class and intra-class complexity.

JHawk produces data complexity (Number of parameters) at the method level and not at the Class level. We were interested in data complexity value at the class level but were unable to convert that raw value by JHawk to class level so we just summed up the number of parameters value for all the methods in the system and treated that as the data complexity. So our value of the data complexity may be slightly high from the actual value of data complexity at the class level.

After all the data was collected, we commenced our analysis. The first thing that interested us was finding out if there is any correlation between system complexities with increasing number of releases.

A general trend that we found is that the system complexity of a software increases with versions with few variation in DataCrow and LaTeXDraw, as shown in Figure 5. The question that cropped up was, what is the reason behind the trend and how can it be explained. Is it because, with version releases, the system is actually becoming more and more complex, or is it because of increase in the size (Number of classes or Number of lines of code) of the system?

We checked the variation for number of classes with the number of versions, and found that the

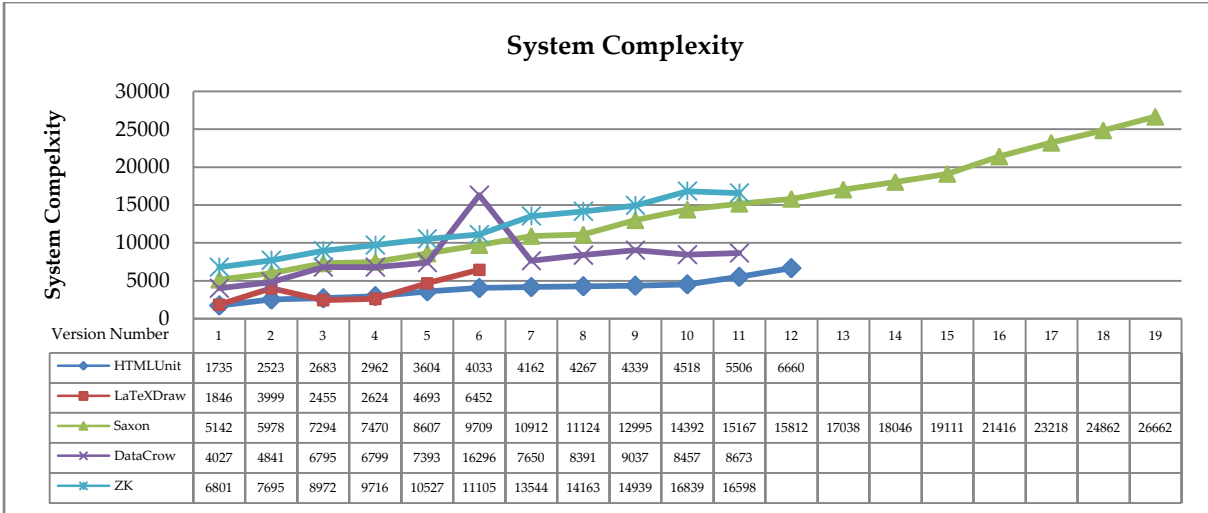


Figure 5: System Complexity graph with their values for all 5 open source Java Software

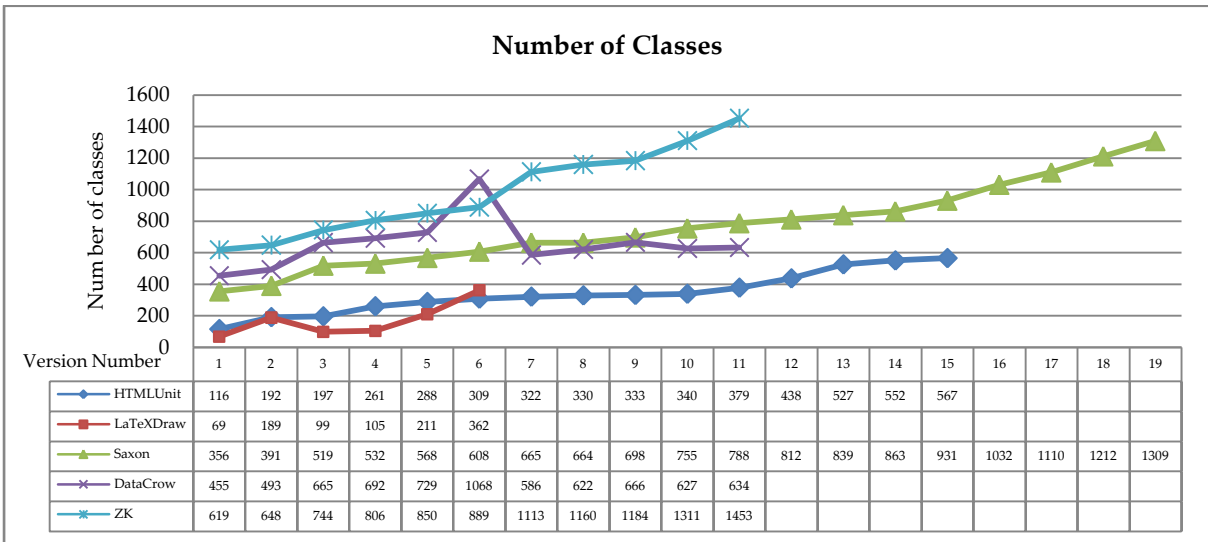


Figure 6: Number of Classes graph with their values for all 5 open source Java Software

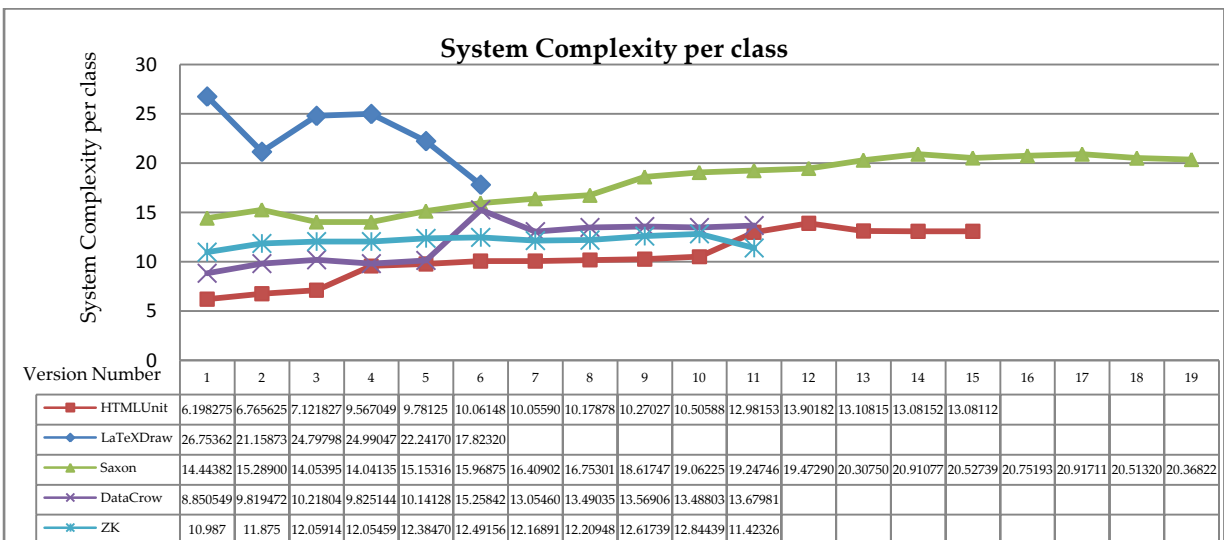


Figure 7: System Complexity per class graph with their values for all 5 open source Java Software

Software Product	Reported post delivery Defects v/s Versions	Reported post delivery v/s System Complexity	Number of Downloads v/s Versions	Reported post delivery Defects v/s Number of Downloads	Reported post delivery Defects per Number of Downloads v/s Versions
ZK	Positive	Positive	Positive	Positive	Positive
DataCrow	Positive	Negative	Positive	Negative	Negative
Saxon	Positive	Positive	Positive	Negative	Positive
LaTeXDraw	Negative	Negative	Positive	Negative	Negative
HTMLUnit	Positive	Positive	Positive	Positive	Negative
<b>Majority</b>	<b>+</b>	<b>+</b>	<b>+</b>	<b>--</b>	<b>--</b>

Table 1: Correlation Table between Variables

complexity increases with because there is an increase in the number of classes (Figure 6) with version releases, with few variations in DataCrow and LaTeXDraw.

The negative deviation in system complexity for DataCrow (Between Point 6 and 7 x-axis, Figure 5) and LaTeXDraw (Between Point 2 and 3 x-axis, Figure 5) can be explained by decrease in number of Classes (Figure 6). This proves that system complexity is related to number of class. By calculated system complexity per class (Figure 7) we found that except LaTeXDraw all other software products shows a slow increase in the values which means that average system complexity per class increases with versions. But on the other hand considering LaTeXDraw the value is decreasing with versions which may symbolises that they are actually introducing more small class with versions.

Table 1 shows the correlation between different variables for different software. Out of 5 open source Java softwares studied, the majority showed a positive correlation between post-delivery defects and versions, and the case for post-delivery defects & system complexity (because system complexity increases with versions, Figure 5) and number of downloads & versions, is also the same. This result shows that with release of new versions, the number of post-delivery defects in the new version will be more as compared to the older versions. The

explanation of this trend may be that with newer version releases, more features are added to the software and the size of the system also increases (Figure 6). Thus, there are more chances of errors being introduced.

We also see the negative correlation (majority) between post-delivery defects & number of downloads and post-delivery defects per number of downloads & versions. This might be explained as: since more defects are introduced with version updates, users prefer to stick to the version they are using, as they may be more concerned about the accuracy of the software than with added features.

According to Awang Abu Bakar and Boughton [1], system complexity and total defects, for various software, has a positive correlation. It's the same trend that we found for different versions of the same software, but with opposite results, in DataCrow and LaTeXDraw. The only explanation for this trend is that perhaps software like ZK, HTMLUnit and Saxon are more concerned with adding more features and functionality to their software with the release of new versions, and then fixing the defects after releasing the software. On the other hand, software like DataCrow and LaTeXDraw believe in getting rid of the defects in their previous versions and then releasing a defect-free old version as a new version with, maybe, few new additional features and functionality.

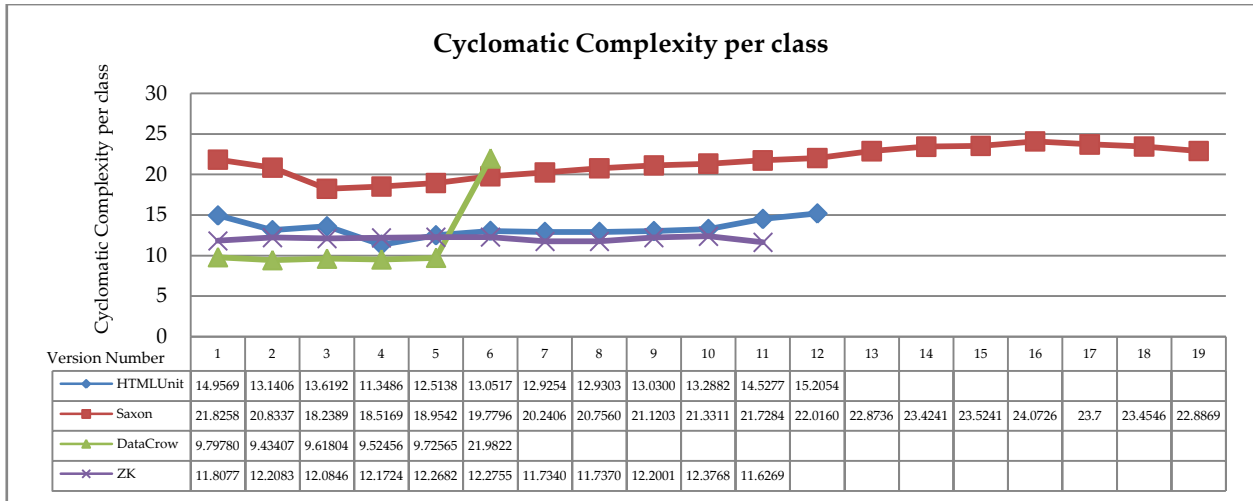


Figure 8: Cyclomatic Complexity per class graph with value for all 5 open source Java Software

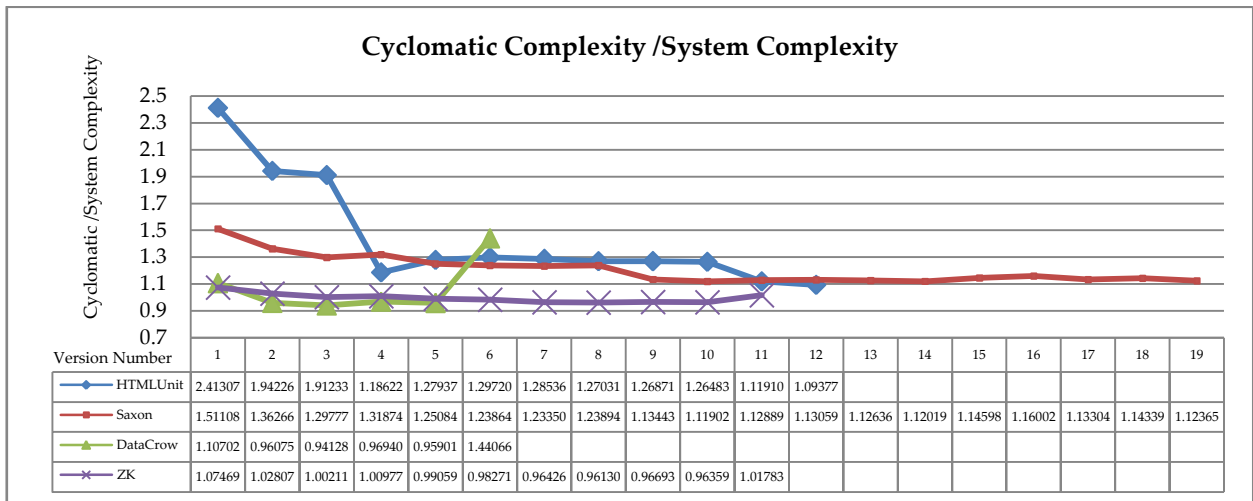


Figure 9: Cyclomatic Complexity divided by System complexity graph with value for all 5 open source Java Software

Figure 8 shows the variation of cyclomatic complexity per class with versions. We calculated cyclomatic complexity using JStyle (metrics-calculating tool). We were, though, unable to calculate the cyclomatic complexity for every version, because JStyle only supports Java files written in Java version JDK 1.4 or below. Continuing the trend by system complexity (Figure 5), Cyclomatic complexity also showed positive correlation for Cyclomatic complexity with the versions. This can easily be explained by seeing the fact that there is increase of the size of the system with version releases. But the interesting fact is that the average Cyclomatic complexity also increases with versions.

Figure 9 is a plot of the Cyclomatic complexity divided by system complexity. The interesting trend that we see in this graph is that in all the system with versions releases either the ratio is near 1 or is tending towards 1. This means that Cyclomatic complexity always counter balances the system complexity or vice-versa in the system. This factor could be a demining feature of the software. It could also be possible that more this factor is near to 1 more stable or mature the software is. More research needs to be done on this.

## 5. Conclusion and Future work

Out of four metrics-calculating tools, we found that JHawk calculates values that are

very close to our manually calculated results and is thus used to gather information, i.e., CBO, Number of parameters, and number of classes for five open source java software products that we studied. Also, JStyle is used to calculate Cyclomatic complexity as it gives values closest to our manually calculated values (Our definition of object-oriented Software).

The main aim of this paper was to use metrics-calculating tools to find correlation between Cyclomatic complexity, system complexity and reported number of defects with different version releases and between system complexity and reported number of defects. We found that there is a positive correlation between the system complexity and the version releases, i.e., with version releases, the system complexity of software increases. We also found that the reported number of post-delivery defects and number of download increases with version releases, but because of increase in post-delivery defects, users may prefer to stick to the old versions, rather than trying new versions. Thus, the number of downloads decreases with the increase in defects. We also found that there is a positive correlation between the system complexity and the reported number of defects.

Future work includes extension of the work, by considering more than five open source Java softwares, to obtain a better picture for the correlation between variables, and to come up with some equation for their correlation.

## 6. Acknowledgements

We would like to thank our colleague Normi Sham Awang Abu Bakar for being such a constant help in validation of tools and reviewing the paper.

## 7. References

[1] Normi S.A.A. Bakar and C. Boughton, "Using a combination of measurement tools to extract metrics from open source projects", (632) Software Engineering and Applications -2008.

[2] D. N. Card and R. L. Glass, "Measuring software design quality" (New Jersey, USA: Prentice Hall, 1990.

[3] S. R. Chidamber and C. F. Kemerer, "A metrics suit for Object Oriented Design", IEEE Transactions on Software Engineering, 1994, 20(6), 476-493.

[4] N.E. Fenton and S.L. Pfleeger, "Software metrics: A rigorous and practical approach", Boston, MA: PWS Publishing Company, 1997.

[5] D. G. Kafura and S.M. Henry, "Software quality metrics based on interconnectivity", *Journal of Systems and Software*, 2(2), 1981, 121-131.

[6] Rudiger Lincke, Jonas Lundberg and Welf Lowe. "Comparing software tools", Proceedings of the 2008 international symposium on Software testing and analysis. Session: Metrics and threads, ACM, 2008, 131-142.

[7] T. J. McCabes, "A complexity measure", IEEE transactions on Software Engineering, 2(4), December 1976, 308-320.

[8] J. W. Paulson, G. Succi and A. Eberlein, "An empirical study of open-source and closed-source software products", IEEE Transaction on Software Engineering, 30(4), April 2004, 246-256.

[9] Y. Zhou and J. David, "open source software reliabilities model: An empirical approach", Proceedings of Open Source Application Space: Fifth Workshop on Open Source Software Engineering (5-WOSSE), St. Louis, MO, USA, 2005.

[10] CKJM: [www.spinellis.gr/sw/ckjm/](http://www.spinellis.gr/sw/ckjm/)

[11]JHawk:  
[www.virtualmachinery.com/jhawkmetrics.htm](http://www.virtualmachinery.com/jhawkmetrics.htm)

[12] JStyle: [www.mmsindia.com/jstyle.html](http://www.mmsindia.com/jstyle.html)

[13] RSM: [www.msquaredtechnologies.com](http://www.msquaredtechnologies.com)

[14] SourceForge.net : sourceforge.net