|  |
| --- |
|  |

**Report on Evaluation of Software Quality and Maintainability of jEdit**

*Anusha Mogili*

*970626-8167*

*mogilianusha127@gmail.com*

*Sravani Bejawada*

*970525-9563*

*shiru2597@gmai.com*

*Manoj Kumar Pallapu*

*970919-2018*

*manojkumarsep1997@gmail.com*

*Ganna Anil*

*960710-7514*

*gannaanil96@gmail.com*

# Abstract

***Software Maintainability, which helps to determine how easily a software system can be maintained. In this paper, the***

***study is done to evaluate the maintainability of jEdit using software quality attributes. A GQM framework is used in this empirical study, which signifies the difficulties in maintaining modules. This paper mainly consists of evaluation of maintainability of software using various metrics that are chosen based on defined goal and quality attributes such as structure, size, complexity, understandability. Obtained results are represented using different statistical and visualization methods.***

***Keywords: Metrics, Maintainability, jEdit, Software Quality.***

**1. Introduction**

Software development life cycle (SDLC) is a process in which different stages are involved for the development, maintenance and understandability of the software project. Maintainability plays a key role in the performance and delivery of a product. [1]

Software Maintainability is defined as at what level the software is easing to modify or maintain. Software maintainability comprises of five sub-characteristics, analysability, changeability, stability, testability and maintainability compliance. Maintainability is an important attribute because it is difficult to access in the software development for reducing cost and time.

Metrics are the indicators of the software quality. Metrics are divided into two categories:  project-based metrics and design-based metrics. Object oriented metrics come under design-based metrics. [2]

Software quality metrics are divided into two types: internal and external attributes. Whereas, Maintainability comes under external attribute. [3]

The main objective of this study is to find the ease of the software while maintaining, by measuring its quality using object-oriented metrics and by comparing the results of source code with different versions of jEdit. The usage of software metrics is mainly implemented on large projects. [4]

The OO metrics are helpful in assessing the quality of the software. The OO metrics are classified as internal and external attributes based on their functionalities. There is different type of metrics, such as WMC, LCOM, CBO, CC, ELOC which are used for evaluating complexity, structure, coupling, cohesion, size etc of the software system.

For example, by using tools like STAN (eclipse plugin), various types of metric values can be extracted. Based on the values generated the quality of the code can be assessed. By evaluating the quality of a software, one can predict the maintainability level of the system. Modules with low cohesion and high coupling leads to bad structure of the code, which effects the quality of the code and results in difficulty while maintenance. [5]

The steps involved in the Project are elaborated and explained by classifying into different sections. In section1, the main goal of the project is explained in detail. In section 2, the suitable research methodology is selected along with data collection and data analysis. In section 3, by using GQM tree the suitable metrics are selected for the questions and represented diagrammatically. In section3.3, the justification of the metrics is given. In section 3.4 mapping of metrics is done. In section 3.3, based on internal and external attributes the scale type selections are done. In section 4, the interpreted results are visualised by considering statistical measures. In section 4.2 reflection points the questions are justified in brief. In section 5, analysis of jEdit maintenance is derived. In section 6, the results are compared with relevant jEdit research papers. In section7, the project is concluded based on briefing about the project.

**2. Research Methodology**

**2.1 Study Type:**

As there are many research methods in empirical study such as experiment, case study, systematic literature review and survey. Case study is selected has the suitable study for this project to investigate the maintainability of the software. As case study gives the detail information in depth and the future work can be predicted using this research method. As experiment doesn’t suit our study type, because we are not imposing a research on a dataset features to a different object for manipulating its behaviour. So, it is irrelevant for this type of study and case study had been opted for better results.

To acquire our goal, we have chosen different platform and jEdit code repository consisting of ten latest versions and imported to generate quantitative data at package level. From analysing the obtained data, the results are compared and depicted with the help of visualization methods and statistical measures i.e. in the form of graphs and tables for better understanding. [6], [7]

**2.2 Information Extraction**

**2.2.1** **Metric Tools Justification**

Open source tools like Eclipse IDE with STAN plugin are used for the collection metrics. The source code of jEdit is clown into the git hub and the code is imported in eclipse

STAN is a metrics plugin of eclipse IDE, it is known as structural analysis of java. STAN is more flexible, easy to use and gives more accurate values for all the metrics. This plugin gives all quantitative metric values along with the Chidamber and Kemerer(CK) metrics, Cyclomatic complexity(CC) metrics, Estimated lines of code(ELOC), Lines of code(LOC) etc. i.e. it gives 5 different types of metric values. [8]

STAN is a useful plugin to generate metric values in different levels, like package, class or member. The main advantage of the STAN plugin, is that it gives visualized and structured graphs with the help of pollution, sandbox features. It has main advantage that the threshold levels will be divided into three different types with different colours like red, green and yellow are obtained. A final report is generated on all the results obtained with pictorial representation and comparing of different metric in STAN plugin and we can download it to for more convenience. This feature helps in deeper analysing and clear understandability of the product is provided. STAN has many features that are undesirable to be found inside one plugin like tree maps hierarchical structure.

In order to achieve our goal, we have selected STAN plugin and generated each metric value. For each attribute we have taken at least two metrics and generated values.

The following are the examples for the STAN plugin can be used to generate these different types of metrics.

For example, metrics like Coupling between object class(CBO), Depth of inheritance tree(DIT), Weighted methods per class(WMC), Lack of cohesion methods(LCOM), Lines of code(LOC), Number of Methods(NOM), Response for Classes(RFC), Instability(I), Afferent coupling(CA), Efferent coupling(CE), and Number of classes(NOC) are measured and data is visualised through Microsoft excel. In addition to build in metrics selection metrics are also measured directly. [9]

Whereas, STAN plugin can’t access more than 500 classes for free, for that we can exclude some of the smaller packages that are present in older versions. For each class, we can generate different metrics and calculate average for each metric to compute at package level. But that not necessary, as STAN plugin can generate metrics at package level.

**2.3 Data Analysis**

Both the qualitative and quantitative data analysis is done in this study. The structured flow graphs are visualised is taken for each metric. Metric values are extracted using metric tools and the results obtained are analysed using visualized methods and statistical measures. [3]

First, the attributes are analysed using scale types such as ratio and absolute based on values generated. The metric values are plotted using line, column and bar graphs depending on the type of values generated and based on internal attributes.

We have obtained column, line and bar graphs randomly. But, with only one exception we have decided to choose the specific graph is: every graph that is used to represent should be easy to understand and look good.

We have mainly used graphs to represent the obtained results for our questions, for better understanding the analysis done.

Modules with difficulty in maintaining is derived based on ratings obtained from calculating mean, average and central tendency to the metric values and tabulated in tabular form.

The statistical measures used in our study is same for all the attributes i.e. average value obtained for each package for every metric is calculated and directly produced by STAN plugin as a overall representing value for that specific module.

On overall analysis throughout the latest ten versions average value, the attribute results will be depended.

Visualisation methods and statistical measures helps us to understand the data produced easily.

Using graphs and tabular form the results are explained in below sections.

In this study, we have used line, column and bar graphs main as visualization methods.

**3. GOAL**

The main goal of the project is to evaluate the maintainability of the jEdit by analysing past releases and compare it with previous articles.

**Goal:** To assess the maintainability of jEdit

* **Specified Task:** 
  + - To analyse the quality of the jEdit software by measuring the attributes like complexity, size, understandability and structure of the code from using relevant metrics.
    - To find which jEdit modules are difficult to maintain for latest ten versions of jEdit software.
* **Platform:** 
  + - jEdit is a software text editor which runs on any operating system.
    - Eclipse IDE is a open source tool selected for importing the source code of jEdit.
    - Structure Analysis (STAN) is a well-known plugin in Eclipse IDE. Which is used to generate metrics for the source code.

**3.1** **GQM TREE**

**GQM tree:** GQM stands for GOAL QUESTION METRIC, this approach is used for extracting metrics to address the given goal. In this approach a set of goal will be determined for suitable purpose. Second thing done in this approach is to form the question to achieve the defined goal. For each framed question relevant metrics are adopted in a quantitative manner to evaluate the maintainability of the jEdit software. The GQM approach is also represented in diagrammatical form using tree structure and tabular form for better representation. Here, the GQM tree helps us to understand the steps to be followed in acquiring the specified goal. [4]

**3.2 SCALE TYPES:**

To the selected entities and attributes scale types are selected based upon mapping. In this study, two scale types are chosen based upon the mapping they are absolute and ratio scale.

**Absolute:** Thisscale is the restricted scale, in which the measurement is done simply counting the number of classes in each entity. [17]

All arithmetic analysis is performed using absolute scale. In this study absolute scale is used for measuring metrics like

For example, counting entities.

**Ratio:** This is most useful scale for measurement, which represents features of other scale types like nominal, ordinal, and interval scales. [17] All arithmetic operations can be performed using ratio scale. The range of ratio scale starts with zero and enhances with equal intervals. For example, time interval, length and temperature.

In this case study, absolute and ratio scale types have been selected based on the situation provided in the goal to address the specified questions and metrics.

****Figure 1: GQM tree



Table 1: Mapping of Entities, Attributes and Measures.

**3.3 Metrics &Scale type Justifications:**

* **McCabe Cyclomatic Complexity (V(G)):** It is used to measure the complexity of the source code. This is calculated by number of distinct paths of the code with respect to the different types of paths, decision points. More the multi directional paths more the complexity.

**CC Metric Justification:** This metric plays a significant role in evaluating maintainability of the software by deriving the complexity of each module. Higher the complexity higher the maintainability. [20]

* **Scale Type Justification:** This metric is used to interpret the complexity, by calculating the number of linearly independent execution paths when compared to other programs. Therefore, absolute is suitable scale type for determining this metric. [12], [24]
* **Weighted methods per class(WMC):** It is the sum of cyclomatic complexity for all number of methods

in a class. Higher the WMC higher the time and more effort required.

**WMC Metric Justification:** With increase in WMC effort required in maintainability also increases with decline in reusability. Using this metric, we get the complexity range values by which we can evaluate the maintainability. [13]

* **Scale Type Justification:** This metric is used   for measuring the complexity of a class by determining the sum of weighted methods. Absolute is chosen as the perfect scale type has it performed all the arithmetic operations and gives the count of elements present in an entity set. Both nominal and ratio scale types are suitable for this metric but they don’t perform all the operations. Nominal scale type doesn’t perform arithmetic operations. Ratio scale type doesn’t count the elements of an entity set. Hence these both scale types are not preferred for this metric. [24]
* **Estimated lines of code(ELOC):** Lines of code measure the exact size of the code by excluding comments, braces, empty spaces. With increase in ELOC, risk of errors also increases and effort required to maintain the code also increases.

**ELOC Metric Justification:** Using this metric, we get the length of the source program, which helps us to evaluate the complexity of the software and the required effort in maintainability. [35]

* **Scale Type Justification:** This   metric is   used for measuring   the cohesion, which measure all the blank lines and spaces in the code. Absolute is preferred as the appropriate scale, justification is given below in NOC.As, ratio does not give all the arithmetic operations.
* **Number of children(NOC):** It is used to measure the number of inherited children from classes within the same package in design structure. Higher the number of children higher the reusability level.

**NOC Metric Justification:** NOC value increases the effort required for the maintainability also increases. This metric helps us in identifying overall class strength present in modules and evaluate complexity of the software.

* **Scale type Justification:** Using this metric, the level of inheritance hierarchy is measured. Both ratio and absolute are used for measuring this metric, but only absolute gives the exact results. As the length, size and area can be determined using this scale. [17]

[14]

* **Number of Methods(NOM):** It used to measure the no of methods, constructors and operations performed, data types defined per class in module. Higher the number of methods higher the size of the class.

**NOM Metric Justification:** Due to increase in NOM the reusability is difficult and excessive no of operations performed in classes leads to larger size with high complexity. Higher the NOM higher the maintenance is required. [34]

* **Scale Type Justification:** As it is used for the counting the number of methods in the class the Absolute scale will be suitable scale type for the Number of methods.
* **Response for classes(RFC):** It is used to measure the no of executed methods based on invoke received directly by class object for one time only. Higher the RFC higher the code size.

**RFC Metric Justification:** It is hard to tend the class behaviour due to mislead of constructors and hierarchy. Higher the NOM, larger the size and declines to maintain. [33]

* **Scale Type Justification:** There will be several invokes of the methods by the classes but it will take it only once even if you call it many times. As they will be at least one call in the class and without the call the method won’t be invoked. Hence, for this absolute scale will be suited as it is counting the no of responses for the classes.
* **Coupling between object class(CBO):** It resembles the dependency of the objects in classes that are coupled. By adopting this metric, we can measure the connectivity between the classes from nested classes and initializers and links among the classes creates difficulty in understanding.

**CBO Metric Justification:** Higher the CBO value difficult in maintenance as dependencies between classes grows complexity level also arises. [36]

* **Scale Type Justification:** This metric is used for measuring the coupling of class, which gives the count of number of classes dependent upon single class. It gives the level of dependency. The other scales are not preferred as the multiple classes are present in each class. In such cases arithmetic operations cannot be performed.
* **Afferent coupling(Ca):** It helps us in Measuring the no of classes that depend on classes within the same package. With increase in Afferent coupling reusability is reduced between interdependency of classes and it is harder to maintain.

**Ca Metric Justification:** By using this metric, we can depict fin(incoming) metric values and higher the coupling lesser the maintenance. [15], [16]

* **Scale Type Justification:** This metric is used for measuring the coupling of a class in an analysed package. Absolute scale type is preferred for this metric which is used to count number of entities present in a package level. In both ratio and nominal scale type are preferred. Due to absence of capability to count the number of entities at package level. [16], [17]
* **Efferent coupling(Ce):** This metrics mainly calculates that total number of classes inside the module which are depended on classes outside the module. Classes with high Ce value are mostly unstable and difficult to maintain.

**Ce Metric Justification:** This metric is mainly adopted to calculate the FOUT (outgoing) metrics to measure the coupling between classes of other package and derive the structural properties of the software. [16], [17]

* **Scale Type Justification:** This metric is used for measuring the number of classes   which depends upon the given class. Absolute scale type is preferred for this metric. The justification is mentioned above in Ca. [16]
* **Lack of cohesion in methods(LOCM):** Used to calculate the cohesion of different methods in classes based on their instances and variable pairs. A class consisting higher value of LCOM means low cohesion. To reduce it, the class should be made into smaller nodes which helps to decrease its count value.

**LCOM Metric Justification:** The relation between methods are obtained by measuring LCOM. If the cohesion value is low then maintain also decreases. Higher the LCOM value lower the cohesion and decrease in maintenance. [19]

* **Scale Type Justification:** This metric is used for measuring the cohesion, the degree of local methods is preferred which are related   to local variables. By using numerous classes measuring was done so other scale types are not suitable. [17]

**WMC**

**Rating**

**CC**

**Rating**

bsh

1

bufferio

1

org.objectweb.asm

2

proto.jeditresource

2

textarea

3

org.objectweb.asm

3

syntax

4

bsh

4

jedit

5

textarea

5

input

6

commands

6

bufferio

7

syntax

7

buffer

8

browser

8

classpath

9

pluginmgr

9

io

10

input

10

* **Depth of inheritance tree:** It resembles the location of class definitions which are extended from the root of class hierarchy level is obtained. As the number of methods and components are increased the deeper it is searched, so, the DIT value also increases as reusability is more and it is difficult to predict the behaviour of class because of more hierarchy. [11] **DIT Metric Justification:** We are selecting this metric, so that we can evaluate the depth of class hierarchy, which in turn helps to measure the complexity and understandability which helps us to evaluate the maintainability. [12]
* **Scale Type Justification:** This is used to measure the understandability metric, to count of number of class in class hierarchy. As absolute gives the measures in detail so only this scale is suitable. [11], [17]
* **Instability(I):** It is used to measure stability of thesoftware product. where stability is measured by calculating the effort to change a package without impacting other packages within the application. (ce/(ce+ca)) ranges between [0,1] if I= 1 indicates most stable category and I=0 indicate most instable category.

**Instability Metric Justification:** understandability is measured using correlation value of instability. Lower the instability higher the maintenance required as understandability is decreased due to un modification of the packages. [18]

* **Scale Type Justification:** This   is used   to measure   the stability within the application without change of impact of package. As the dependencies are to be calculate so, absolute is chosen as the appropriate scale type compared to other scale types. [18]

**4. Results**

**4.1 Answers to GQM questions**

**Q1. Which jEdit modules have relatively high complexity?**

For measuring the complexity of jEdit modules, we have selected WMC and CC metrics. By observing the generated metric values, we have chosen the top 10 packages with highest complexity and which are common in the latest ten versions of jEdit. From the ratings of table 2, we can see that bsh package has the highest complexity metric value of WMC along with org.objectweb.asm, textarea and syntax. Whereas, the other packages like jedit, input, bufferio, buffer, classpath, io are listed in top ten for consisting high values of complexity for WMC metric while bufferio, proto.jeditresource, org.objectweb.asm, bsh has the high complexity values for CC metric along with commands, syntax, textarea and browsers are in top ten. Packages like input, puginmgr and jedit are found to be in CC metric of top ten packages with high complexity values for only few versions of jEdit software. For WMC metric, the bsh package has the highest complexity value around 51.77 and for CC metric it is bufferio package with 6.26 value for all latest ten versions of jEdit.

Table 2: Represents highest complexity packages according to their ratings

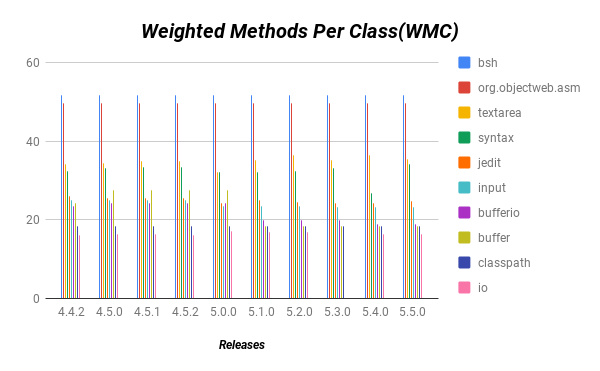


Figure2: Represents WMC metric values for jEdit.

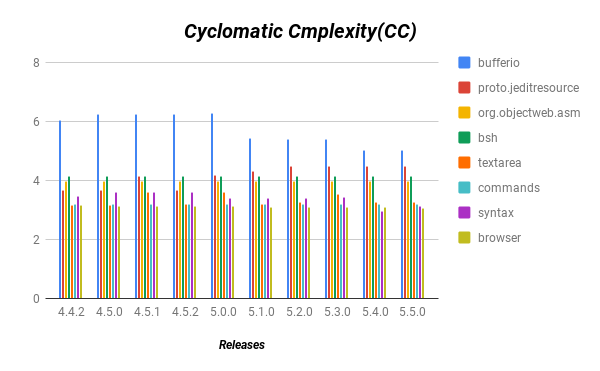


Figure 3: Represents CC metric values for jEdit.

Packages with highest complexity is derived from the values greater than 3.5 for CC metric and values that are more than 25 for WMC metric. To conclude the packages with highest complexity are bufferio, proto. jeditresource, jedit, bsh, org.objectweb.asm, textarea.

**Q2. Which jEdit modules have regularly modified?**

For obtaining frequently updated packages, we have selected two metrics such as, ELOC and NOM. Increasing in the values of LOC and number of methods leads to regularly changed modules. From below table 3, packages like bsh, jedit, and gui are the most frequently modified packages with values above 15000. Whereas, package like textarea with value around 11000 is not more frequently modified but consists of changes on regular basis. Other packages such as options, browser, pluginmgr and search are less frequently modified with value less than 10000. The packages with highest LOC values are also found in NOM metric with more no of methods for bsh, jedit, textarea with values between 8.5 and 12.5. Org.objectweb.asm has the highest NOM value with 12.5 which is more frequently updated and io has the least NOM value with 7.0. The below table shows the regularly modified package of jedit.

|  |  |
| --- | --- |
| **Packages** | **Frequently Modified** |
| bsh | More Frequently |
| jedit | More Frequently |
| gui | More Frequently |
| org.objectweb.asm | More Frequently |
| textarea | Frequently |
| syntax | Frequently |
| options | Less Frequently |
| io | Less Frequently |

Table 3: Represents top modules that are frequently modified.

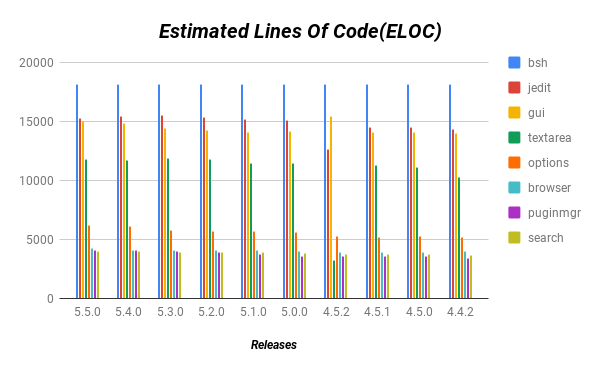


Figure 4: Represents ELOC metric values for jEdit.

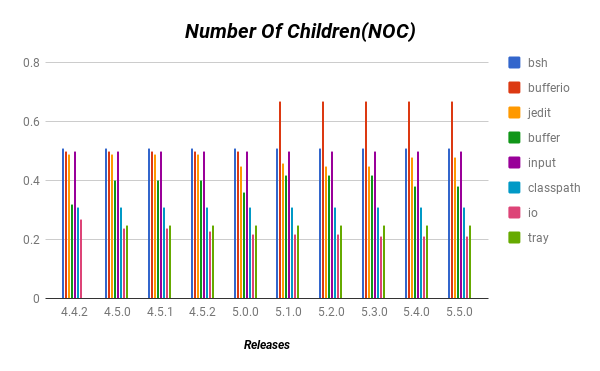


Figure 5: Represents NOC metric values for jEdit.

**Q3. Which jEdit module have relatively high coupling?**

We have selected CBO, Ca and Ce metrics for collecting the top ten values with relatively high coupling.  From the table 4, below, it shows us that packages like jedit, browser are listed top among them with CBO metric values around 15 and 9. Other packages like io, bsh, pluginmgr, textarea, search has CBO metric values ranging from 5-8. Whereas, packages like bufferio, print and syntax have the least coupling CBO metric values in top ten and appear in few versions of jEdit. For other coupling metrics like Ca and Ce, jedit package has the highest value. Packages with highest Ce values are gui and browser around 50 & 45. Likewise, packages with highest value of Ca are buffer and gui with above 60. From this we can conclude that the top packages with high coupling are jedit, browser, gui, buffer, io.

|  |
| --- |
| **High Coupling** |
| jedit |
| browser |
| io |
| gui |
| buffer |
| textarea |

Table 4: Represent top modules that possess High coupling.

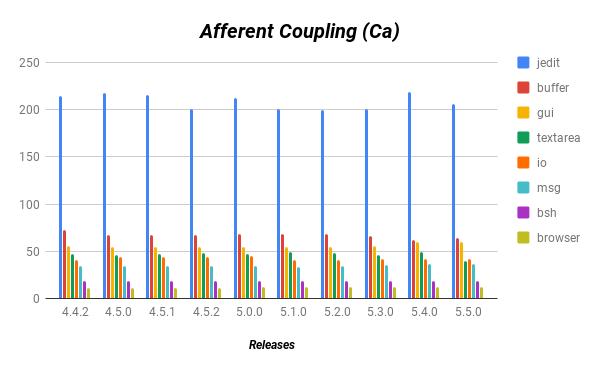


Figure 6: Represents Ca metric values for jEdit.

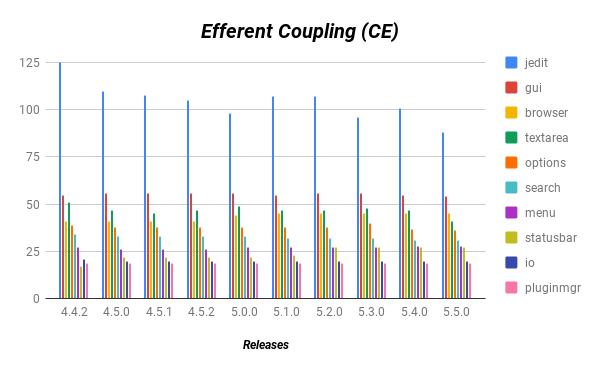


Figure 7: Represents Ce metric values for jEdit.

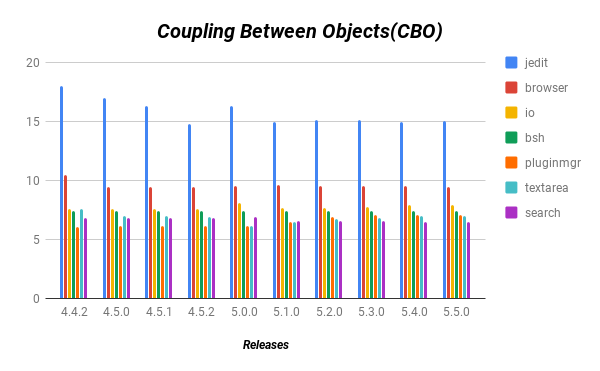


Figure 8: Represents CBO metric values for jEdit.

**Q4. Which jEdit modules have low cohesion?**

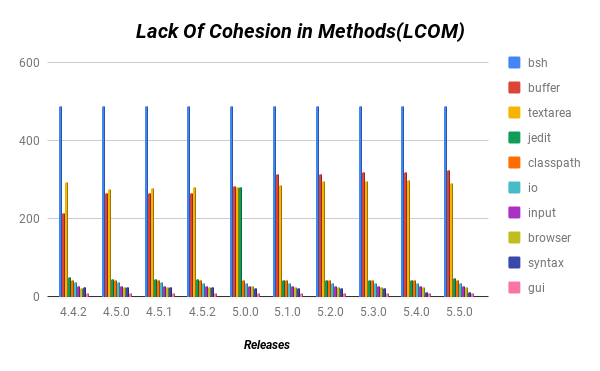


Figure 9: Represents LCOM metric values for jEdit.

LCOM metric had been chosen to measure the cohesion modules. We have collected the top ten highest value of LCOM, which possess low cohesion. From the table 5, below, package with highest LCOM is bsh with value above 450. Other packages such as, buffer, textarea and jEdit has the LCOM value which exceeds 250. Some of the packages with moderate change in numbers and made into the top ten packages of least cohesion are classpath, io, input, browser are below 50 LCOM metric value. The last with least cohesion packages in top ten are browser, syntax and gui with values below 25. This leads us to the conclusion with top packages in low cohesion are: bsh, buffer, textarea, jedit, classpath.

|  |
| --- |
| **Low Cohesion** |
| bsh |
| buffer |
| textarea |
| jedit |
| classpath |

Table 5: Represent top modules that possess low cohesion.

**Q5: Which jEdit modules have relatively poor code structure?**

Low cohesion and high coupling leads to poor code structure. Modules with low cohesion are often found to be highly complexed. Whereas, modules with high coupling are relatively hard to maintain. From the below table 6, packages like jedit, bsh, browser, buffer has relatively poor code structure. Packages that are slightly changed in numbers but have least code structure are io and textarea. Other packages with least code structure are gui and classpath. To conclude with the top packages of poor code structure are jedit, bsh, browser, buffer.

|  |  |  |
| --- | --- | --- |
| **Coupling** | **Cohesion** | **Poor Code Structure** |
| jedit | bsh | Very Poor |
| browser | buffer | Very Poor |
| bsh | browser | Very Poor |
| io | textarea | Poor |
| gui | jedit | Less Poor |
| buffer | classpath | Less Poor |

Table 6: Represent top modules that possess poor code structure.

**Q6. Which of the jEdit modules have a poor level of understandability?**

To measure the understandability of the code, two metrics have been studied: Instability(I) and Depth of inheritance tree (DIT) metrics. From this we have collected metric values for top ten instability and DIT values. By observing the generated values, we founded that top eight packages are found to be same in lowest instability and highest DIT values for all the ten latest versions. so, we have chosen those values to interpret our results. As the DIT values increases, packages with low level instability are stable with more inter dependencies, which makes it hard to modify and this leads to poor understandability. The below table 7, depicts, DIT with highest value packages and instability with lowest packages from top ten latest releases. We can see from the below figures that packages like org.objectweb.asm, buffer, msg, bsh have least instability values and appeared in every version of jEdit. Other packages like jedit, io, syntax, input have relatively low instability levels. Instability values ranges from 0 to 0.5 which are likely to be stable. Whereas packages like msg, bufferio, reflect, bsh has high DIT metric value and other packages like browser and options has relatively low levels of high DIT metric values. DIT values ranges from 1 to 2.25 signifying the level of depth associated with complexity and hard to understand. Finally, we can conclude that the packages with poor degree of understandability are: org.objectweb.asm, buffer, msg, bsh, bufferio, jedit, reflect, collection.

|  |  |
| --- | --- |
| **DIT** | **Instability** |
| msg | org.objectweb.asm |
| bufferio | buffer |
| reflect | msg |
| bsh | bsh |
| collection | jedit |
| classpath | io |
| browser | syntax |
| options | input |
| tray | textarea |
| textarea | gui |
| buffer | visitors |

Table 7: Represent top ten modules for DIT and Instability metrics.

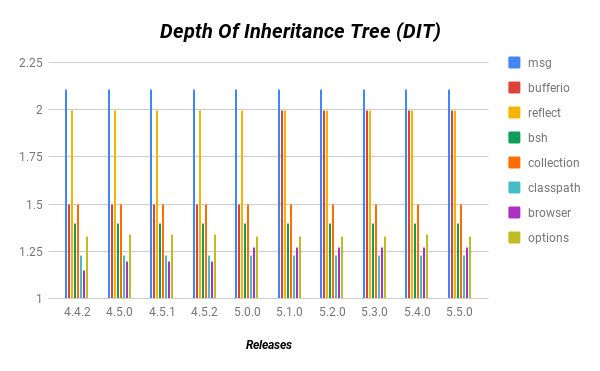


Figure 10: Represents DIT metric values for jEdit.

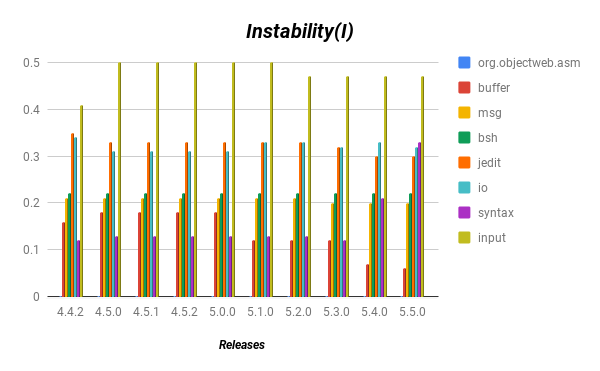


Figure 11: Represents Instability metric values for jEdit.

**7Q. What is the impact caused by evolution of each release on internal attributes and how is it related to maintainability of the code?**

To explain the impact of evolution by each release on internal attributes, we had selected packages based on their complexity levels and formed a sample to use it as a representation for whole packages of jEdit software. Further, the complexity levels had been categorized into high, moderate and low complexity levels. We have considered WMC metric to measure the complexity levels of the package. Based on generated metric values, we have observed that the packages with high complexity level are ranging from 20 to 55. Other packages with moderate level of complexity would exceed 10 and the remaining packages have low complexity levels below 10 compared to other packages.

We have selected randomly four packages from three different categories mentioned above, i.e. twelve packages out of 29 packages. For these twelve packages we have taken different types of metrics with respect to their attributes and obtained results to interpret their impact and maintainability level.

|  |  |
| --- | --- |
| Complexity Level | Modules |
| Highly complexed | Bsh, org.objectweb.asm, textarea, syntax, jedit, input. |
| Moderate | Buffer, classpath, io, commands, browser, proto. jeditresource, search, gui, pluginmgr, print |
| Less complexed | Bufferset, menu, options, collection, help, indent, tray, datatransfer, msg, statusbar, reflect, visitors. |

Table 8: Represents different types of complexity

* **jEdit Complexity**

From the fig 12, we can observe that packages bsh and org.objectweb.asm has consistently highest values of complexity in all ten versions. Whereas packages with moderate complexity level like textarea and buffer had decreased at versions 5.0 to 5.2.0 and increased its complexity levels as 5.4.0 versions. Other packages with low complexity levels like bufferset and options remained same throughout all the ten versions. Packages with highest complexity are difficult to maintain. Finally, we can conclude that the impact caused by evolution of releases on complexity had slight changes and affects the maintainability level of jEdit software.

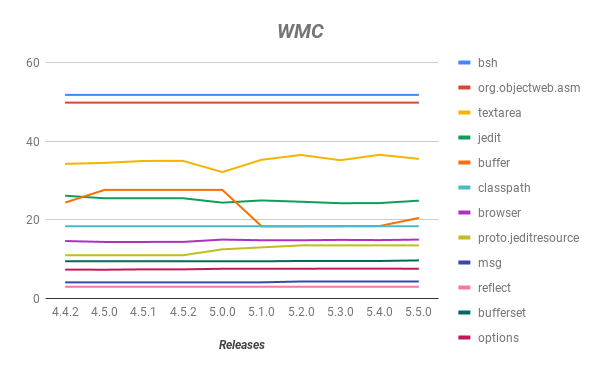


Figure 12: Represents WMC metric values for jEdit.

* **Size of jEdit Modules**

Size play a major role in evolution of a software, from initial phase to latest released versions there will be huge difference. In jEdit modules size can be divided into three categories, i.e. larger modules with above 15000 LOC, modules with above 10000 LOC and smaller modules with less than 10000 LOC values. Packages with above 15000 LOC are larger in size and these packages are also high in level of complexity and those packages are bsh, jedit. Other packages with medium size of LOC such as, textarea and options has increased from version to version in regular intervals. Whereas complexity for textarea has high and options has decreased its complexity level as low. Next, the packages with least complexity such as msg, bufferset has a smaller LOC and are at same level for all latest versions. To obtain accurate results we had compared our results using RFC metric and obtained the results as textarea, bsh, jedit packages are with highest values and modules like msg and bufferset are found to be with least values. So, we can conclude that, size has impact caused by evolution of releases by increasing in size. From the fig 13 below, we can depict that packages with larger size has a high complexity and these are the packages that are difficult to maintain. Higher the complexity Lesser the maintainability.

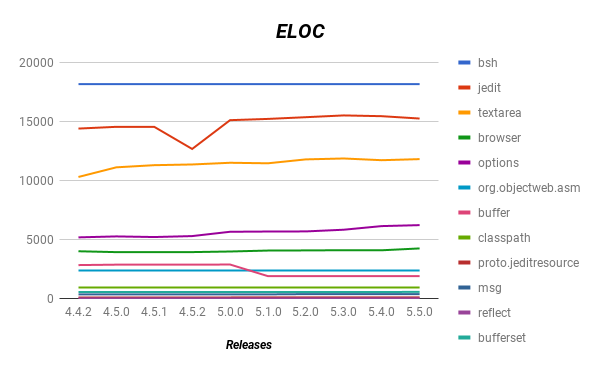


Figure 13: Represents ELOC metric values for jEdit.

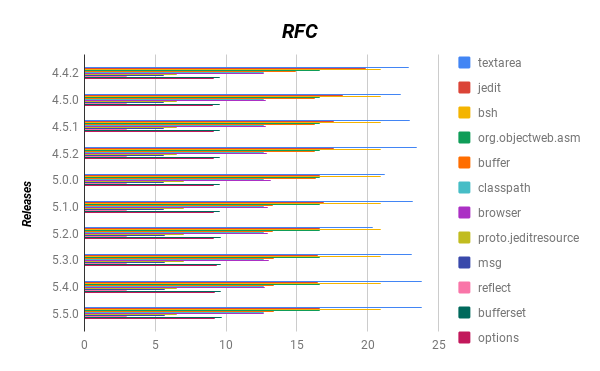


Figure 14: Represents RFC metric values for jEdit.

* **Structure of jEdit Modules**

Structure of the code can be decided using coupling and cohesion factors. Highest coupling and lowest cohesion can result in poor code structure. To obtain jEdit software code quality we have taken CBO and LCOM metrics and generated values for each of its package.

The fig 15, shows us that the CBO values are are high for jedit and browser, even their LOC and complexity levels are higher. Packages like bsh and textarea are ranging from 5 to 8 with high CBO values. Other packages such as msg, reflect, bufferset packages are same throughout the ten versions, even by comparing to complexity and size. But one thing that we noticed in CBO graph is that all the packages are decreasing from initial version to latest version and some are remaining constant like classpath, options modules. To finish the analysis of CBO, the impact caused by evolution of releases on coupling had decreased significantly with respect to size and complexity.

From the fig 16, we can depict that every package has the constant level of cohesion throughout the ten versions except for buffer and text area. Package bsh has the highest LCOM value, with high complexity level and larger in size.  Whereas buffer value for complexity and size decreases gradually but cohesion increases with releases of versions, this states that poor cohesion is obtained.

Packages like textarea are larger in size with high complexity level and has high LCOM value, which means it has low cohesion value. Other packages like msg, org.objectweb.asm, reflect, options, bufferset has a consistent value for all ten versions with their size and complexity levels are also low (below 3000 LOC & 15 WMC values). To conclude jEdit modules has a minimal impact by evolution of releases. Packages which has moderate complexity are the packages that are affected by poor cohesion, these packages are difficult to maintain.

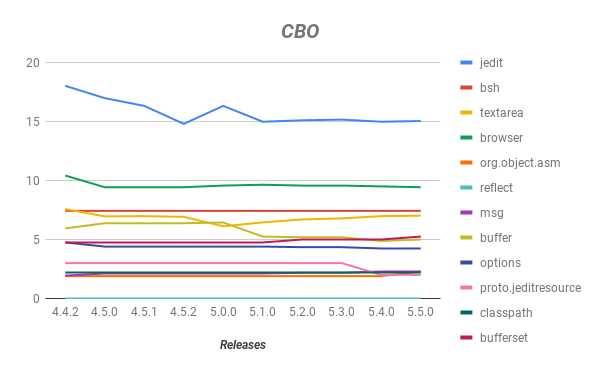


Figure 15: Represents CBO metric values for jEdit.

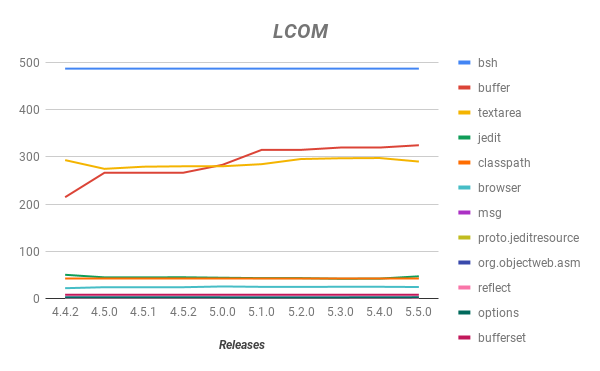


Figure 16: Represents LCOM metric values for jEdit.

By equating the results of coupling and cohesion presented above, we had noticed that jEdit modules have a poor code structure. As LCOM and CBO values are decreased for the latest versions, when compared to initial versions. By this we can conclude that structure had a impact by evolution of packages.

* **jEdit Understandability Level**

The level of understandability for a software plays significant role. Understandability can be measured by stability of modules or depth of inheritance of the package. The below fig 17, represents DIT graph for measuring understandability. Packages like msg and reflect have highest DIR values above 2.0. These packages have low complexity, smaller in size, poor code structure and low level of understandability. Package like textarea has increased at version 4.5 and decreased at 5.1 versions and has high DIT value at latest version i.e. 5.5, when compared to medium and small packages. Next highest DIT values is acquired by bsh and options packages, which are ranging from 1.0 to 2.0. But understandability of packages is increased for packages like jedit, proto. jeditresource which are having high complexity levels and larger in size. Whereas, packages like buffer and bufferset has better understandability levels but their complexity levels are moderate. By this observation we can say that understandability had affected due to evolution of releases on moderate and small packages. Other packages have remained constant throughout all ten versions with slightly increased in DIT values when compared to older versions. Low level of understandability means higher the maintenance required.

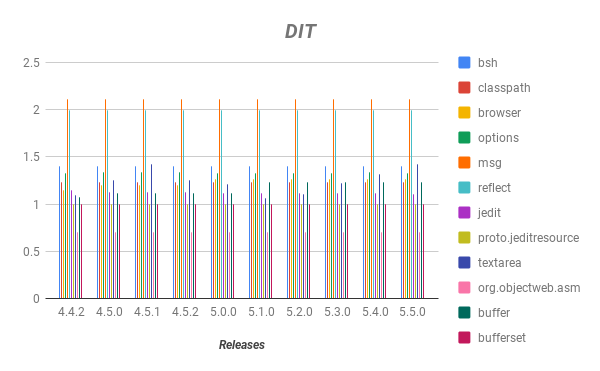


Figure 17: Represents DIT metric values for jEdit.

These are the packages with highest values in each attribute for randomly selected 12 packages. From observing the below table 9, we can say that packages like bsh, jedit, textarea are the hardest modules to maintain.



Table 9: Represent modules that are hard to maintain.

According to the results obtained, every attribute is interrelated. As size of the code increase its complexity is highly increased with poor structure and low level of understandability.  But, in some cases it actual it doesn't mean that package with high complexity need not to be in larger size or poor understandability. Impact caused is lesser on the jEdit for all attributes. Code complexity had been reduced, less change in size for all ten versions and there is poor understandability level but it didn't get affected by the code quality as structure of jEdit didn’t get affected by evolution of releases.

The impact caused by evolution of releases is correlated with maintainability as, higher the complexity, larger the size, poor level of understanding and poor quality of structure leads to difficulty in maintaining.

From above analysis, bsh, jedit and textarea packages are the difficult ones to maintain.

**4.2 Reflecting Points in Our Study**

**R1.** **Does larger size of the product module affect its complexity?**

Increase in size of the modules tend to become more complex. Larger the size, higher the complexity of the code. To evaluate the complexity of the larger modules we have selected WMC, NOM metric. From the values generated in fig 19, we have Chosen the top three packages of jEdit with high complexity such as textarea, bsh, org.objectweb.asm and founded that the three packages are also in larger code size of all latest ten versions of jEdit. Fig 18, shows the WMC metric graph with highest complexity values ranging from 35 to 51.  Higher the complexity higher the risk of maintainability.

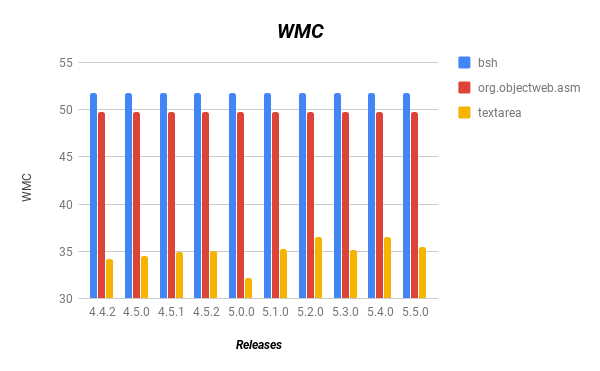


Figure 18: Represents WMC metric values for jEdit modules.

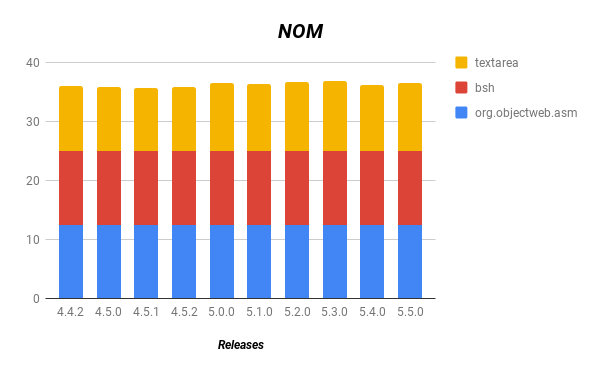


Figure 19: Represents NOM metric values for jEdit modules

**R2. Which of the product modules will be more difficult to maintain? And Why?**

Modules with increase in size leads to higher complexity, poor structure and understandability of the code.

Modules having low cohesion and high coupling leads to poor code structure. Highly unstable packages lead to poor level of understandability. Modules with these features are more difficult to maintain. From the analysis of jEdit software, we think bsh and jedit packages are difficult to maintain, because these two packages are large in size with poor cohesion, high coupling, higher level of depth of inheritance and highly complexed in all latest ten versions of jEdit modules. To be short, jedit and bsh packages are having large size, highly complexed, poor structure with poor understandability. To conclude these packages are more difficulty to maintain. In fig 20, represents NOC and Instability graphs of jedit and bsh packages. These figures prove that jedit and bsh packages have increased in size and poor understandability. As number of children increase, the understandability of code decreases due to more inter dependencies.

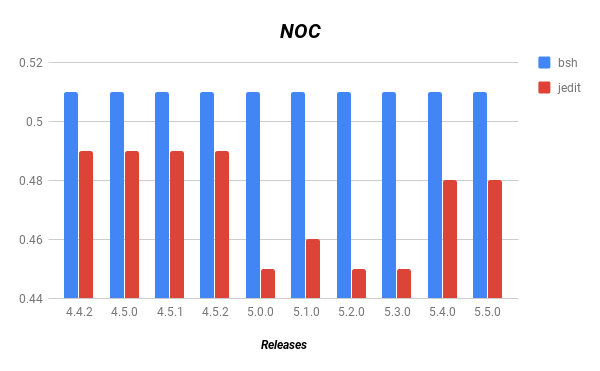


Figure 20: Represents NOC metric values for jEdit modules.

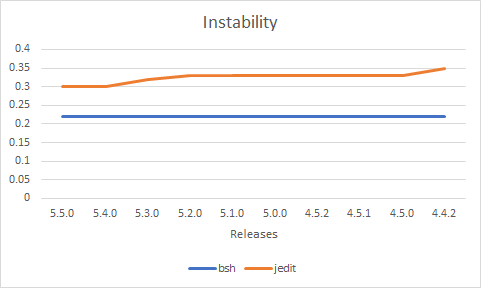


Figure 21: Represents Instability metric values for jEdit modules.

**R3. How can we improve the maintainability of these modules?**

The above modules possess high complexity, large size, poor structure and understandability. So, in order to improve the maintainability of these modules we need to optimize the code. This can be done by breaking complex code into smaller execution modules and reducing number of dependencies between classes to decrease coupling. Removing inter dependencies from dividing the classes based on unused variables to increase cohesion.  Size can be decreased by coding in shorter unit of code i.e. using less methods, constructors and fewer decision points while avoiding duplication of source code such as removing hierarchy between nested classes, reducing execution of code repeated times. By doing this the code can be optimized to a certain level and makes it to easily understand with a better code structure. These are some of the methods to improve the maintainability of these modules.

**R4. Which of the product modules possess poor code structure and how are the difficult to understand?**

Modules with low cohesion and high coupling possess poor code structure. As the structure of the modules is poor, the understandability level of those modules will also decrease with more dependencies.

We have taken top five packages with poor code structure and poor level of understandability, by generating metric values of latest ten versions of jEdit software. The table 10 below, shows that in list of top five packages, at most three packages with poor code structure are difficult to understand. This proves that modules that possess poor code structure are difficult to understand. The below figures represent LCOM, CBO and Instability metric graphs of the three packages bsh, jedit and buffer. The packages have poor structure and understandability as CBO values exceeds LCOM and Instability values.  As LCOM increase cohesion decreases with high coupling leads to poor structure and with least instability value, code couldn't be modified as it is hard to understand.

|  |  |
| --- | --- |
| Structure | Understandability |
| jedit | org.objectweb.asm |
| browser | buffer |
| bsh | msg |
| io | bsh |
| buffer | jedit |

Table 10: Represent top modules that have poor structure and understandability.

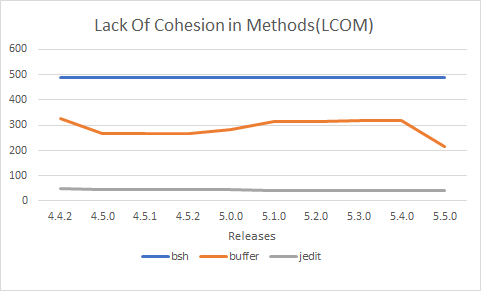


Figure 22: Represents LCOM metric values for jEdit modules.

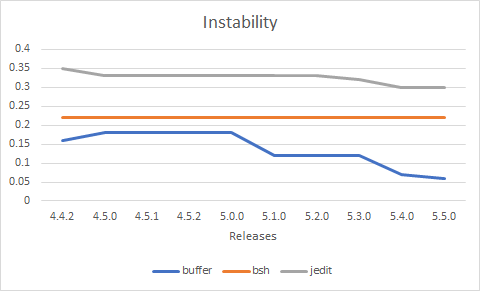


Figure 23: Represents Instability metric values for jEdit modules.

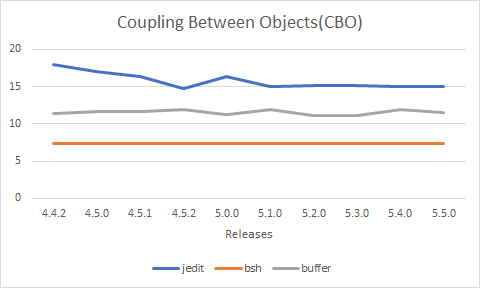


Figure 24: Represents Instability metric values for jEdit modules.

**R5. Which of the product modules are frequently increased in size and does they possess poor/ better code structure?**

By studying the last ten packages the modules that were frequently updated are larger in size and some of them possess high complexity, poor structure and low level of understandability. We have selected top five packages in each and every attribute by generating and comparing metric values. The results depicted that five out of three packages such as bsh, jedit, gui, for all latest ten versions of JEdit were frequently updated and larger in size with poor code structure. For this analysis we had selected LOC, WMC, CBO, LCOM metrics as appropriate and observed the results. Frequently updated modules have a poor code structure on basis of high coupling with low cohesion values were discovered. The below figures represent the three packages that has increase in size and poor code structure. Increase in size of the modules depicts that they were frequently updated. To conclude, the packages with frequent modification can tend to have poor code structure. Whereas, poor code structure does not only depend on regular change in its size but it also depends on other attributes such as complexity, understandability and maintainability.

|  |  |  |  |
| --- | --- | --- | --- |
| Packages | Size | Packages | Structure |
| bsh | More Frequently | jedit | Very Poor |
| jedit | More Frequently | browser | Very Poor |
| gui | More Frequently | bsh | Very Poor |
| org.objectweb.asm | More Frequently | gui | Poor |
| textarea | Frequently | buffer | Less Poor |

Table 11: Represents the top packages that are modified frequently and has a poor structure.

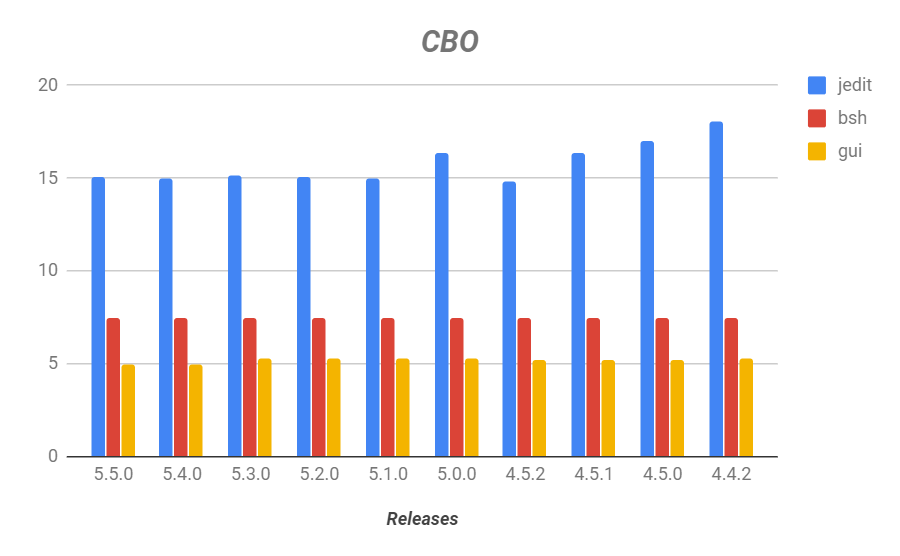
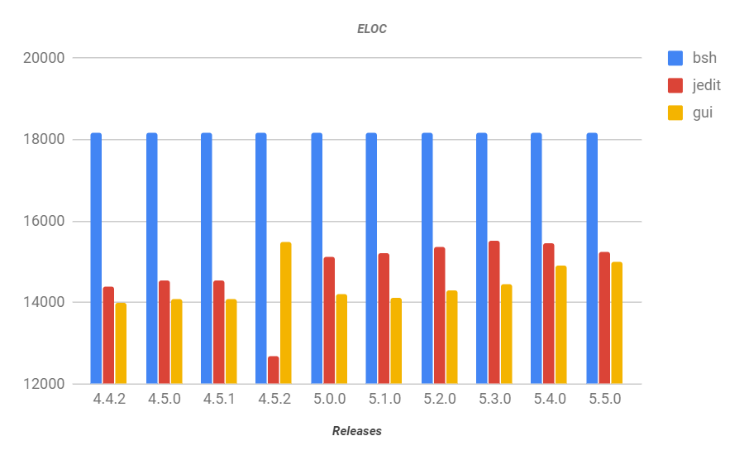


Figure 25: Represents CBO metric values for jEdit modules.



|  |  |  |  |
| --- | --- | --- | --- |
| Size | Structure | Complexity | Understandability |
| bsh | jedit | bufferio | org.objectweb.asm |
| jedit | browser | proto.jeditresource | buffer |
| gui | bsh | org.objectweb.asm | msg |
| org.objectweb.asm | io | bsh | bsh |
| textarea | gui | textarea | jedit |
| syntax | buffer | jedit | collection |

Figure 26: Represents ELOC metric values for jEdit modules.

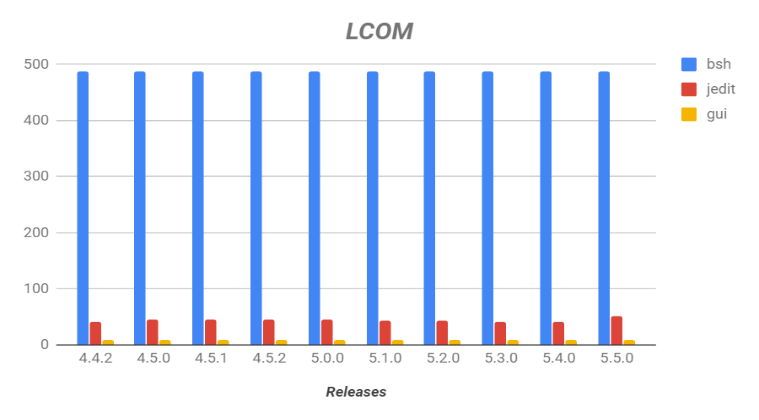


Figure 27: Represents LCOM metric values for jEdit modules.

Table 12: Represents top packages in every attribute.

**5. Analysis of jEdit Maintainability**

The specified goal of this study is to find the modules that are difficult to maintain. To acquire this goal, we have answered questions based on four different attributes and by analysing each of them we obtained the following results: bsh and jedit are the difficult packages to maintain because of their high complexity and they are in larger size, poor structure and low level of understandability. Next, is the package textarea with high complexity and increased in size is at top three in both attribute and it has low LCOM value, which is ranked least in the table. Even understandability is at good terms by ranking at edge of the table.  Whereas, buffer package consists of smaller size, low coupling, but has high complexity, low cohesion, low understandability which makes it difficult to maintain. Next, org.object.web package which has neutral values for all attributes and listed in top packages in all versions is also difficult to maintain. From these results, we can conclude that these modules are difficult to maintain based on generated metric values, but it doesn't indicate that these modules have poor quality or jEdit has poor maintainability, because these packages are an assumption based on metric values, and metrics we have chosen can be used for measuring multiple attributes, an accurate measure for single attribute is not given. Every factor counts in determining how difficult they are to maintain. These packages are difficult to maintain based on one factor i.e. metric generated values only.

**Q. How does other internal attributes such as structure, complexity and understandability of code get affected by size?**

Increase in size of the code also effect complexity of the code and leads to poor structure of the code with low level of understandability. For example, we have taken the top six packages of latest ten versions of JEdit software for size, structure, complexity and understandability attributes. In table 12, we can see that Packages with highest LOC metric values like bsh and jedit have high complexity, poor structure and low level of understandability.

But this may not be same for every package because in other cases, top packages with large size like org.objectweb.asm has only high complexity and poor understandability and packages like buffer with poor structure and understandability doesn't have to be in large size and highly complexed. So, we can conclude that some packages with large size can tend to have high complexity, poor structure and low level of understandability.

**6. Discussions**

**6.1 Related Work**

**In paper [1],** the author used different metrics to assess quality of the design for the jEdit versions. They used the metric 1.3.6 eclipse plugin for calculating the metric values of three versions of jEdit software. They calculated the size, C&K and others metrics for the versions i.e. 3.2, 4.0, 4.31 and by observing the metric values we noticed, that there is a huge difference between the version 4.0 to 4.3.1 in the size of code and complexity while between the version 3.2 and 4.0 the difference is minimal. The values of the LCOM shows that classes of the packages should be redesigned and all the packages have very high instability which means it has poor internal structure. By evaluating the results of the metrics there is no significant development in the design of jEdit. [29]

**In Paper[2]**, author has used a various approaches to discover the changes in maintainability of 41 versions of the jEdit. Mainly, they used three approaches in these i.e. manually inspecting the source code, maintainability index of jEdit and comparing the results of both manual and MI values. In the manual approach, the graph of the package and classes has been gradually increasing for every module. Complexity had been increased in the classes obviously for all 41 versions. We can obtain maintainability index using Hastead value formula and it is totally different from the manual approach as the MI value is gradually increasing for the version 2.42 to 3.0, then it is being gradually decreased for the other versions and it will be almost similar for the last 10 versions. It proposes the method to assess the maintainability of the large-scale software and it is used to assess jEdit modules. [32]

**In Paper [3],** they have used the martin’s cohesion metrics and proposed metrics to calculate the values for the four different java software applications such as camel, Tomcat, jEdit and jHotDraw. They have explored the relation between the cohesion and the maintenance effort of the packages which are only developed in java. The results can be different for the other languages like C++ etc. By this study, they found that packages with higher cohesion can be easily maintained. [31]

**Comparing our results with the papers**, in paper [1] they have analysed jEdit modules and based on results founded that most of the methods and classes should be redesigned. the quality of the jEdit for all the modules hasn’t been changed significantly, same as our study results i.e. for the starting versions is high and then for later versions it is minimally decreased. In another paper [2], they have used different approaches to measure the maintainability of the jEdit modules, which is same as our study we should also find the Maintainability of the jEdit versions by taking it as the reference. In article [3], they used the four open source java applications to show the correlation between the cohesion and maintainability of packages which is useful to our results to find the maintainability with the cohesion. They have concluded with low cohesion and higher maintainability required, but in our study, we had difficulty in maintaining modules.

**6.2 Reflections:**

By doing this project, we have learnt how to calculate the various metrics for any new software in the market and evaluate its maintainability. We have also learned how to extract metrics from the Eclipse IDE and with the help of STAN plugin. Now, we can assess the various attributes of the software and estimate the flexibility and how ease to use the application. We can control the software development process by giving the inputs such as, in what area the code should be optimized for the better performance of the software product.  We have also learnt to adopt GQM framework and measure the quality of a software. We also had an opportunity to learn about different statistical measures and visualization methods and to represent it neatly. This course mainly helps us in learning detail about different types of metric and how to assess the quality of a software. It helps a lot for our future work while testing a software product  and gains us a huge opportunity in future companies.

**7. Conclusion**

This study mainly is focused on analysing jEdit modules and to find out the modules that are difficult to maintain. For this study case study case study is chosen as an appropriate method. Goal Question Metric paradigm was used to define questions and their suitable metrics which satisfies the study of our goal. Suitable Metrics are mapped to the relevant questions based on quality attributes and entities that are related to maintainability of a software. For extraction of metrics we have used, STAN plugin in eclipse IDE. The results we have obtained are, the packages like bsh, jedit and textarea are difficult to maintain. By overall analysis, we think that jEdit require less effort to maintain. We want to notify that the maintainability of software does not depend on only metric values generated but also their other factors plays a significant role like threshold value. More research and analysis should be done to know the accurate quality of jEdit software and we can know the benefits of using object-oriented metric more clearly.

**8. References**

1. M. Asadi and H. Rashidi, “A Model for Object-Oriented

Software Maintainability Measurement,” *International Journal of Intelligent Systems and Applications; Hong Kong*, vol. 8, no. 1, p. 60, Jan. 2016.

1. G. Gousios, E. Kalliamvakou, and D. Spinellis, “Measuring Developer Contribution from Software Repository Data,” in *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, New York, NY, USA, 2008, pp. 129–132.
2. L. Ping, “A Quantitative Approach to Software Maintainability Prediction,” in *2010 International Forum on Information Technology and Applications*, 2010, vol. 1, pp. 105–108.
3. R. Malhotra and A. Chug, “Software Maintainability:

Systematic Literature Review and Current Trends,” *Int. J.*

*Soft. Eng. Knowl. Eng.*, vol. 26, no. 08, pp. 1221–1253, Oct. 2016.

1. M.-C. Lee and T. Chang, “Software Measurement and

Software Metrics in Software Quality,” *International Journal of Software Engineering and Its Applications*, vol. 7, no. 4, p. 21, 2013.

1. V. R. Basili and H. D. Rombach, “The TAME project: towards improvement-oriented software environments,” *IEEE Transactions on Software Engineering*, vol. 14, no. 6, pp. 758–773, Jun. 1988.
2. S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian, “Selecting Empirical Methods for Software Engineering Research,” in *Guide to Advanced Empirical Software Engineering*, Springer, London, 2008, pp. 285–311.
3. “STAN - Structure Analysis for Java,” *Eclipse Plugins, Bundles and Products - Eclipse Marketplace*. [Online]. Available: https://marketplace.eclipse.org/content/stanstructure-analysis-java. [Accessed: 20-Apr-2018].
4. “MetricsReloaded:” *JetBrains Plugin Repository*. [Online]. Available: https://plugins.jetbrains.com/plugin/93metricsreloaded. [Accessed: 20-Apr-2018].
5. “JHawk - the Java metrics tool - Product Overview.” [Online]. Available:

http://www.virtualmachinery.com/jhawkprod.htm. [Accessed: 20-Apr-2018].

1. S. Sarkar, A. C. Kak, and G. M. Rama, “Metrics for Measuring the Quality of Modularization of Large-Scale Object-Oriented Software,” *IEEE Transactions on Software Engineering*, vol. 34, no. 5, pp. 700–720, Sep. 2008.
2. N. Fenton and J. Bieman, *Software Metrics: A Rigorous and Practical Approach, Third Edition*. CRC Press, 2014. [13] A. Kaur, K. Kaur, and K. Pathak, “A proposed new model for maintainability index of open source software,” in *Infocom Technologies and Optimization Proceedings of 3rd International Conference on Reliability*, 2014, pp. 1–6.
3. M. I. Sarwar, W. Tanveer, I. Sarwar, and W. Mahmood, “A comparative study of MI tools: Defining the Roadmap to MI tools standardization,” in *2008 IEEE International Multitopic Conference*, 2008, pp. 379–385.
4. G. Concas, M. Marchesi, A. Murgia, S. Pinna, and R.

Tonelli, “Assessing traditional and new metrics for objectoriented systems,” 2010, pp. 24–31.

1. Y. Ayalew and K. Mguni, “An Assessment of Changeability of Open Source Software,” *Computer and Information Science*, vol. 6, no. 3, p. 68, May 2013.
2. R. D. W. Neal, “The Assignment of Scale to ObjectOriented Software Measures,” Jun. 1997.
3. “Instability (I = Ce / (Ca + Ce)) (slow): Robert C. Martin’s Instability.” [Online]. Available:

http://staff.unak.is/andy/staticanalysis0809/metrics/i.html. [Accessed: 28-May-2018].

1. S. K. Dubey and A. Rana, “Assessment of Maintainability Metrics for Object-oriented Software System,” *SIGSOFT Softw. Eng. Notes*, vol. 36, no. 5, pp. 1–7, Sep. 2011.
2. J. P. Ostberg and S. Wagner, “On Automatically Collectable

Metrics for Software Maintainability Evaluation,” in *2014*

*Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement*, 2014, pp. 32–37.

1. K. Hashim, “A Software Maintainability Attributes Model,” *Malaysian Journal of Computer Science*, vol. 9, no. 2, pp.

92–97, Dec. 1996.

1. J. Al Dallal, “Object-oriented class maintainability prediction using internal quality attributes,” *Information and Software Technology*, vol. 55, no. 11, pp. 2028–2048, Nov. 2013.
2. S. R. Chidamber and C. F. Kemerer, “A metrics suite for object oriented design,” *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, Jun. 1994. [24] L. Rosenberg, “Applying and Interpreting Object Oriented Metrics,” presented at the Software Technology Conference, 1998.
3. K. D. Welker, P. W. Oman, and G. G. Atkinson, “Development and Application of an Automated Source Code Maintainability Index,” *Journal of Software Maintenance: Research and Practice*, vol. 9, no. 3, pp. 127– 159.
4. A. Kaur, K. Kaur, and K. Pathak, “Software maintainability prediction by data mining of software code metrics,” in *2014 International Conference on Data Mining and Intelligent Computing (ICDMIC)*, 2014, pp. 1–6.
5. B. Swaminathan and K. Jain, “Implementing the Lean Concepts of Continuous Improvement and Flow on an Agile Software Development Project: An Industrial Case Study,” in *2012 Agile India*, 2012, pp. 10–19.
6. A. Abuasad and I. M. Alsmadi, “Evaluating the correlation between software defect and design coupling metrics,” in *2012 International Conference on Computer, Information and Telecommunication Systems (CITS)*, 2012, pp. 1–5.
7. “Metrics in Assessing the Quality and Evolution of jEdit,” *ResearchGate*. [Online]. Available: https://www.researchgate.net/publication/258367965\_Metric s\_in\_Assessing\_the\_Quality\_and\_Evolution\_of\_jEdit. [Accessed: 28-May-2018].
8. M. I. Hoque *et al.*, “An Empirical Study on Refactoring Activity,” p. 11.
9. Albattah, Waleed. "An empirical investigation of the correlation between Package-level cohesion and maintenance effort." *International Journal of Advanced Computer Science and Applications (IJACSA)*.
10. A. Molnar and S. Motogna, “Discovering Maintainability Changes in Large Software Systems,” in Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement, New York, NY, USA, 2017, pp. 8
11. <http://staff.unak.is/andy/staticanalysis0809/metrics/rfc.html>
12. <http://support.objecteering.com/objecteering6.1/help/us/metrics/metrics_in_detail/number_of_methods.htm>
13. <http://staff.unak.is/andy/staticanalysis0809/metrics/loc.html>.
14. http://www.virtualmachinery.com/jhawkmetricsclass.htm