Software Metrics Project - Report on Jabref

Koyyada Sai Pranav
990429-9311
saky19@student.bth.se

Vishruta Dochibhotla
990807-1609
vido19@student.bth.se

***Abstract*—Maintainability defines the progression and cost associated with an entity making it the primary concern in organisations. This report evaluates the maintainability of Jabref, an open sourced cross-platform reference and citation manager written in java. The last 10 releases of Jabref are analysed to identify the packages difficult to maintain using a Goal Question Metric(GQM) based empirical study. The results demonstrate packages with poor maintainability followed by validation of the findings in comparison with maintainability index.**

***Index Terms*—Software Metrics, Software Maintainability, GQM framework, Jabref**

I. Introduction

Software metrics are quantifiable or symbolic representation of attributes and process associated with an entity.[5] With consistent evaluation and verified process, these metrics translate to the overall quality of the entity.[6] Software metrics are critical resources for quality control in infrastructure susceptible to continuous change.[7]

The constant requirements change and improvement of entity demands exceptional maintainability. Maintainability is a measure of potential of an entity to adapt and retain changes. Quality of an entity considerably relies on its maintainability. An entity is not expected to scale or improve with appalling maintainability.[8] Maintenance defines majority of the project cost over its span. Hence, it is a point of attention and importance in organisations.[9]

Maintainability is an indirect measure and hence classified as an external attribute. It can be evaluated based on the internal attributes of the entity. Internal attributes exhibit varied relationships with maintainability thus commenting on the ease of changes or restoration of the system.[10][11]

This report evaluates the maintainability of Jabref, an open-sourced, cross-platform reference and citations manager written in Java with a Goal Question Metric(GQM) based empirical study. The last ten releases of Jabref are examined and analysed as a Case study to identify the modules that would be more difficult to maintain. The relevant internal attributes to measure maintainability are captured from Systematic Literature Review(SLR) following the guidelines proposed by Kitchenham et al.[12] Measurable data from a set of standard metric extraction tools were visualised and analysed using appropriate methods to interpret the maintainability of the entity. The primary findings include packages of Jabref with lower maintainability.

The Research Methodology of this report is presented in section 2. Section 3 cites the Tools used for metric extraction from the ten releases of Jabref. Visualisation techniques and statistical measures are explained in Section 4. The GQM Tree is showcased in Section 5 followed by its Metrics description and justification in Section 6. The Scale types of these metrics are addressed in Section 7. The complete analysis and findings of the report are documented in Results in Section 8. The Discussion in Section 9 covers a brief examination of related work followed by the reflection points of the study. Threats to validity are presented in Section 10. Conclusion in Section 11 closes the report by briefly summarising the work.

II. Research Methodology

Case Study is an empirical investigation of contemporary phenomenon in real life context from multiple sources of evidence. They support an exploratory rationale with flexible design and are generalisable theoritically. Given the rationale of this study being exploratory and confined to a case of Jabref, Case Study is chosen as the appropriate Research Methodology. [13][14]

Case Study encapsulates diverse set of data specific to the case thus improving the findings and conclusions. Also its preference for realism over controlled factors and support for many variables compliments the overall structure of the study. [13][14]

Experiments are discarded because of their Explanatory rationale and fixed design. Also the variables in the current study are diverse with little or no control. Surveys are not considered as they provide a descriptive analysis and relies on opinions thus making it unhealthy for current study.

The case study of this report is grounded in data with relevant visualisations and statistical measures to derive and back the findings.

III. Metric Extraction Tools

This section briefs about the tools used in extracting the relevant metrics from all the ten versions of Jabref

- **Metrics Reloaded:** Metrics Reloaded is an IntelliJ IDE plugin which has abundant metrics for project analysis. The plugin itself is an Open Source Software (OSS) and

once installed provides a menu option under 'Analysis' with the name 'Calculate Metrics'. The evaluations can be performed at various levels such as project, module, file and the tool even offers custom scope for evaluation. A wide variety of metrics can be extracted using this tool, for example, Chidamber-Kemerer, Class count etc. are some metrics that can be extracted using this tool.
We used this tool to extract the Chidamber-Kemerer (CK) metrics and the Martin's metrics for the project under analysis.[24]

- **Lizard:** Lizard is a python package licensed under MIT and was first released on September 16, 2013. From then, it has a total of 90 releases with the latest one on Jan 5, 2020. This package is a code analyzer that works on most of the programming languages like Java, C/C++, JavaScript, Python, Ruby etc. and includes metrics like cyclometric complexity and so on. In our study, we used this to get the complexity metrics and also the NCLOC(Non Commented Lines Of Code).[25]

- **Prest:** Prest is an open-source tool for software metric extraction, analysis and defect prediction. This tool was developed by the Software Research Laboratory(Softlab), Computer Science Department of Bogazici University, Turkey. It was developed with the aim to have a tool which could not only extract metrics but also have a model which predicts defect-prone areas in a project. Prest supports extraction from 5 different languages and is capable of extracting 28 code attributes. Halstead Difficulty of Jabref was extracted using this tool. [26]

- **JHawk:** JHawk is an open source Java code analyser at different levels of abstraction. Its initial release ten years ago, saw the first IDE integration and report generation using in multiple formats. JHawk was used to extract Maintainability Index of the modules. [27]

- **UCINET and Structure 101:** UCINET: It is a software package developed by Analytic Technologies. It is aimed at analysis of social network data and is embeded with the NetDraw network visualization tool. Structure 101: Structure 101 is a visualization tool that helps in visualizing the structure and architecture of the software. It can be integrated with most development environments as a plugin. It also can be used with Jenkins, Maven and SonarQube. The above two tools were used to get the Eigen Centrality values. [28][29]

IV. Visualisation and Statistical measures

The metrics of all ten releases of Jabref extracted from tools enlisted in Section 3 were visualised with the following techniques:

- **Column Charts:** Column charts are rectangular box representation of a Y coordinate value associated with a category or component mapped on the X coordinate. In this report, Column charts are used to compare the individual metric value of each module of the Jabref system.
- **Line Charts:** Line charts map coordinate values as markers from a continuous function connected with a straight line. In this report, Line charts are used to analyse the trend of individual metric value of a package across different releases of Jabref.
- **Tables:** Tables are row and column structured data mapping for swift access. They are used to illustrate different formats and class of data in this report.
- **NetDraw:** Netdraw is a visualisation tool for social network data.[15][30] This report uses NetDraw to understand the dependency network and centrality among the modules of Jabref.

The Statistical methods used to analyse the data are:

- **Mean:** Mean defines the centrality of a numeric set N. In this report, few metrics extracted at the class level are translated into package level using mean.

V. Goal Question Metric (GQM) Tree

Goal Question Metrics (GQM) is a paradigm for interpretation and analysis of collected data oriented with a goal[1].

GQM tree defines a measurement model decomposed into 3 layers:

- Conceptual(Goal): A goal is bounded with an entity to comment, analyse or improve its characteristics.
- Operational(Question): A question encompasses the attributes of the entity relevant to address a defined goal.
- Quantitative(Metrics): Metrics are quantifiable measure of an attribute to derive numeric or symbolic representations for analysis and answering the associated question.

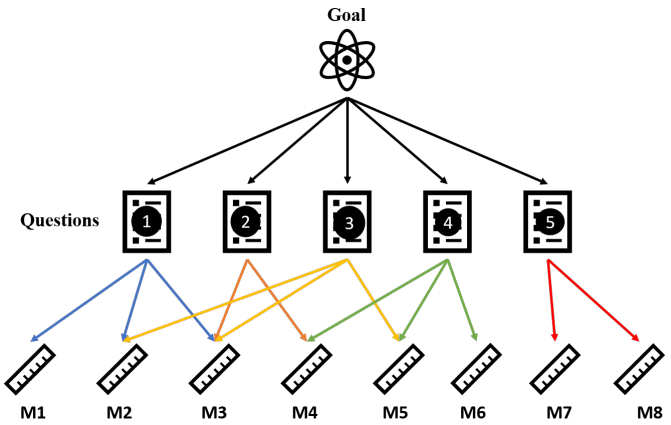

Fig 1: Template of GQM Tree

GQM tree translates goal to questions answerable with set of metrics[1][2]. Fig 1 depicts a template for GQM tree

with 3 layer decomposition. In the current report, a GQM based empirical study is practiced to evaluate maintainability of JabRef. Table 1 shows a tabular representation of GQM measure model specific to this report.

Table 1: GQM for Jabref

Goal		
To identify JabRef modules that would be difficult to maintain		
Questions		
Code Structure and Complexity	Q1	Which modules of Jabref are cardinal for the system?
	Q2	Which modules of Jabref have relatively poor coupling?
	Q3	Which modules of Jabref have relatively poor cohesion?
	Q4	Which modules of Jabref contain complex code ?
Code Understandability	Q5	What is the magnitude of readability and understandability of each Jabref module?
Code Size	Q6	Which modules of Jabref witnessed an increased and frequent change in code size across the releases?
Metrics		
Q1	M1	Eigen Centrality
Q2	M2	Afferent Coupling (Ca)
	M3	Efferent Coupling (Ce)
	M4	Coupling Between Objects (CBO)
Q3	M5	Lack of Cohesion of Methods (LCOM)
Q4	M6	Cyclomatic Complexity (CC)
	M7	Weighted Methods per Class (WMC)
Q5	M8	Halstead Difficulty (HD)
	M9	Depth of Inheritance (DIT)
	M10	Comment Ratio (CR)
Q6	M11	Non Commented Lines of Code (NCLOC)
	M12	Number of Methods (NOM)

The goal of this study is to identify the modules of Jabref that would be difficult to maintain. This goal is refined into 6 questions constructed around 4 internal attributes of the entity Jabref answerable with 12 metrics.

VI. Metrics Description and Justification

Table 2 shows the metric categorisation based on the internal attributes they address. Combination of metrics are used to answer questions associated with one category to address the goal better.

Table 2: Metric categorisation

ID	Metric	Internal Attribute
M1	Eigen Centrality	Centrality
M2	Afferent coupling (Ca)	Coupling
M3	Efferent coupling (Ce)	
M4	Coupling Between Objects (CBO)	
M5	Lack of Cohesion of Methods (LCOM)	Cohesion
M6	Cyclomatic Complexity (CC)	Complexity
M7	Weighted Methods per Class (WMC)	
M8	Halstead Difficulty (HD)	Understandability
M9	Depth of Inheritance (DIT)	
M10	Comment Ratio (CR)	
M11	Non Commented Lined Of Code (NCLOC)	Size
M12	Number of Methods (NOM)	

A brief description and justification for application of each metric is enlisted below:

A. *Centrality Metrics*

- Eigen Centrality: Eigen Centrality is a Social Network Analysis(SNA) metric measuring the criticality of a module in the network/System. It is defined as the number of connected modules to the current module with individual connectivities taken into account. Thus, a module connected to another module with many connections holds higher centrality. [15]

  Justification: Eigen Centrality identifies the critical modules in the system thus commenting on the overall code structure and inter-module dependencies.

B. *Coupling Metrics*

- Afferent Coupling (Ca): Afferent Coupling is defined as the measure of total number of external classes coupled to the classes of a package due to incoming coupling. Alternatively, it is the number of classes in another package depending on the classes in the current package. [16]

  Justification: Afferent coupling is inversely proportional to Instability of the class. Instability is negatively correlated to maintainability. Therefore, stability of class and maintainability of a class increases with increase in Afferent coupling. [16][19]

- Efferent Coupling (Ce): Efferent Coupling is defined as the measure of total number of external classes coupled to the classes of a package due to outgoing coupling. Alternatively, it is the number of classes in another

package that the classes of current package depend upon. [16]

Justification: Efferent coupling is directly proportional to Instability of the class. Instability is negatively correlated to maintainability. Therefore, stability of class and maintainability of a class decreases with increase in Efferent coupling. [16][19]

- Coupling Between Objects (CBO): Coupling Between Objects is defined as the dependency of object of current class with an external object belonging to another class. It is measured as the count of these external classes the current class is coupled to. [17] Statistical mean of CBO of each class within the package was considered to translate it to package level abstraction in the current study.

  Justification: CBO shows negative correlation with maintainability as a result of increased complexity and dependencies between classes. Hence if CBO is low then Maintainability of the class is high.[17]

C. *Cohesion Metrics*

- Lack of Cohesion of Methods (LCOM): Lack of Cohesion of Methods is measure of method associativity with the local variables of the class collectively realising a task. LCOM count is incremented by one provided there is no association or reference to the local variables, else is decremented.[17] Statistical mean of LCOM of each class within the package was considered to translate it to package level abstraction in the current study.

  Justification: Encapsulation is directly proportional to the cohesion. Thus higher cohesion means better Code Structure. Projects with good structures are easier to maintain. Hence, Lower the value of LCOM, better the maintainability.[17]

D. *Complexity Metrics*

- Cyclomatic Complexity (CC): Cyclomatic Complexity is defined as the count of independent linear paths in program control graph. It is a positively correlated to code complexity.[18] Statistical mean of CC of each class within the package was considered to translate it to package level abstraction in the current study.

  Justification: Increased complexity demands greater effort and resources to understand and contribute to the code. This behaviour makes code resistant to change thus declining maintainability. Therefore, lower Cyclomatic Complexity is desired for better maintainability. [18][19]

- Weighted Methods per Class (WMC): Weighted Methods per Class is the sum of CC of all local methods. The complexity of the entity is defined by CC and number of methods in the class.[17] Statistical mean of WMC of each class within the package was considered to translate it to package level abstraction in the current study.

  Justification: Methods are defined to realise a unique sequence of task. With more number of methods having complex code, the effort and resources to manage the system grow exponentially. It barricades easy revisions and restorations to the entity hence dampening its maintainability. Therefore, WMC should be maintained low to keep maintainability of the system high. [17]

E. *Understandability Metrics*

- Comment Ratio: Comment ratio is the density of comments in the source code of the entity measure as the ratio between Commented Lines of Code (CLOC) to Total Lines of Code (TLOC).[18]

  Justification: Comment density contributes to the overall readability and understandability of the code thus reducing the complexity overheads and allowing easy revisions. Hence, CR positively correlates to maintainability.[18]

- Halstead Difficulty (HD) : Halstead difficulty is a reflection of the code obscurity and measured as ratio of Total number of operands to the number of distinct operands times half of distinct operators in the source code of the entity or module. Operands are variables and operators are functions working on the variables.[18][20]

  Justification: Source code with large set of operators and operands are difficult to read and understand. Hence, Lower HD is essential for good maintainability.[18][20]

- Depth of Inheritance tree (DIT): Depth of inheritance defines the path between the root and node class. It is the count of ancestors a class has inherited from. Therefore, DIT of root class is 0 and minimum value of DIT for any of the node class is 1. [17] Statistical mean of DIT of each class within the package was considered to translate it to package level abstraction in the current study

  Justification: With more ancestors to track, any given class generates a massive complexity overhead thus declining the understandability. Code change might be implemented from the root if the structure demands. This behaviour makes classes resistant to change. Hence, DIT is inversly proportional to maintainability. [17]

F. Size Metrics

- Non Commented Lines of Code (NCLOC): Non Commented Lines of Code is the portion of source code ignoring the comments. NCLOC is a major contributor to the code size.[18][19]

  Justification: Complexity and understandability overheads are increased with growing code size making it difficult to maintain. Therefore lower NCLOC contributes to good maintainability.[18][19]

- Number Of Methods (NOM): Number Of Methods counts all the defined methods within a class irrespective of the access level. Every individual method reflects a specific task.[21]

  Justification: The size of the code increases with more functionality thus affecting its complexity and understandability. Therefore if total NOM is high, then the maintainability is low. [21]

VII. Scale Type

Table 3: Scale Types of each chosen metric

Internal Attribute	ID	Metric	Scale Type
Centrality	M1	Eigen Centrality	Ratio
Coupling	M2	Afferent coupling (Ca)	Absolute
	M3	Efferent coupling (Ce)	Absolute
	M4	Coupling Between Objects (CBO)	Absolute
Cohesion	M5	Lack of Cohesion of Methods (LCOM)	Absolute
Complexity	M6	Cyclomatic Complexity (CC)	Absolute
	M7	Weighted Methods per Class (WMC)	Absolute
Understandability	M8	Halstead Difficulty (HD)	Ratio
	M9	Depth of Inheritance (DIT)	Absolute
	M10	Comment Ratio (CR)	Ratio
Size	M11	Non Commented Lined Of Code (NCLOC)	Absolute
	M12	Number of Methods (NOM)	Absolute

Table 3 shows the scale type of each chosen metric to evaluate the maintainability of Jabref. A brief description and justification of using these scales is provided below:

1) Ratio: Ratio scale inherits the characteristics of nominal, ordinal and interval scale with an additional zero element where the scale starts and increases at equal intervals.It supports all arithmetic analysis.[22] The following metrics use the Ratio scale:
   a) **EC:** EC is a criticality measure of modules with connectivity of neighbours taken into account.[15] The inclusion of weight distribution concepts inclines the EC as a metric with Ratio scale.

   b) **HD:** HD is an understandability reflection of source code measured as a ratio of total number of operands to the number of distinct operands times half of distinct operators in the source code of the entity or module.[18][20] Since the metric itself explicitly calls out for ratio, it is included under a ratio scale.

   c) **CR:** CR is an understandability reflection of source code measured as the ratio of CLOC to TLOC. It begins with 0 and increases in fixed intervals.[18][22] Therefore a ratio scale is apt for CR metric.

2) Absolute: Absolute is a restrictive count scale used by counting elements in the entity. It supports all arithmetic analysis. [22] The following metrics use the Absolute scale:
   a) **Ca:** Ca is a count of incoming coupling links at package level therefore absolute scale is suitable for Ca.[16][19]

   b) **Ce:** Ce is a count of outgoing coupling links at package level therefore absolute scale is suitable for Ca.[16][19]

   c) **CBO:** CBO is a count of coupled external classes.[17] Therefore at class level, CBO is measured with an absolute scale. However, taking the package level abstraction for this report into account, Statistical mean of CBO of each class within the package was considered to translate it to package level abstraction.

   d) **LCOM:** LCOM is a method-variable associativity measure incremented if method has no reference to the variable.[17] Since its a count value, absolute scale is ideal to measure this metric. However, taking the package level abstraction for this report into account, Statistical mean of LCOM of each class within the package was considered to translate it to package level abstraction.

e) **CC:** CC is a complexity measure describing the count of linear independent paths in the program flow.[18] Hence, absolute scale is suitable. However, taking the package level abstraction for this report into account, Statistical mean of CC of each class within the package was considered to translate it to package level abstraction.

f) **WMC:** WMC is measured as the sum of CC of all local methods. [17] Since CC belongs to absolute scale and all arithmetic operations are supported over it, WMC inherently is classified as a metric with absolute scale. However, taking the package level abstraction for this report into account, Statistical mean of WMC of each class within the package was considered to translate it to package level abstraction.

g) **DIT:** DIT counts the ancestral nodes for current class. Since this metric involve counting elements of an entity we use absolute scale to measure it. [17]

h) **NCLOC:** NCLOC is the count of lines of source code ignoring the comments. [18][19] Therefore absolute scale is used to measure it.

i) **NOM:** NOM is the count of methods irrespective of the access type. [21] Since this metric involves counting elements of an entity, absolute scale is suitable to measure it.

VIII. Results

Jabref is a open sourced, cross-platform reference and citation manager written in java. A GQM based empirical study is practiced to evaluate its maintainability. The last ten releases of Jabref are examined and analysed as a Case study to identify the modules that would be more difficult to maintain. The metrics identified relevant to address this goal are documented in section 7 of this report.

Jabref was built around Swing Framework during its initial release in 2003. In 2017, the development emphasis shifted towards migration of framework from Swing to JAVAFX1. Until version 3.8 Jabref used Swing framework. With release version 4, Jabref successfully migrated from Swing to JAVAFX1.[23] Alongside the framework upgrade, numerous changes were implemented throughout the last 10 releases. These changes include addition, updating, removal or merging of packages between releases.

Fig 2 shows the comparison of Number of root level packages across the last 10 releases of Jabref. The fall in the number of packages in every release suggest improved code structuring through the versions.

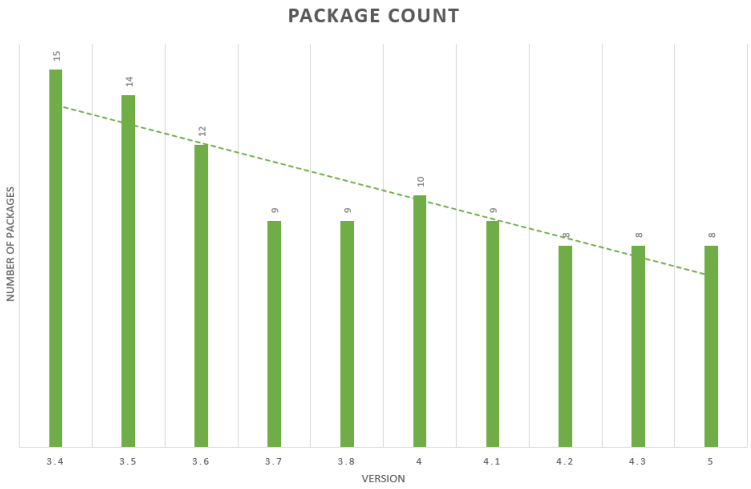

Fig 2: Number of root level packages across 10 releases of Jabref

Fig 3 shows the comparison of TLOC across the last 10 releases of Jabref. There is a visible growth in code size reflecting upon the added functionalities and refinements in every release.

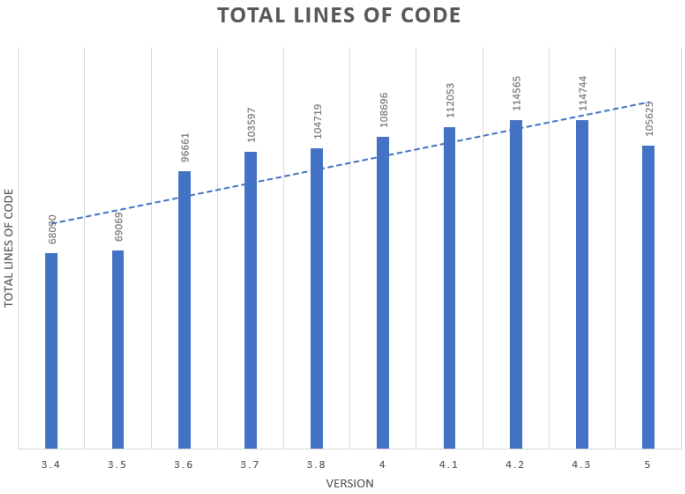

Fig 3: TLOC across the last 10 releases of Jabref

A Total of 19 unique packages were identified at the root level of Jabref in org.jabref (sf.jabref until version 3.8) across the last 10 releases as candidate modules for this GQM empirical study. The list of these packages is presented below:

- jabref.architecture
- jabref.cli
- jabref.gui
- jabref.logic
- jabref.migrations
- jabref.model
- jabref.preferences
- jabref.styletester

- jabref.pdfimport
- jabref.shared
- jabref.collab
- jabref.event
- jabref.external
- jabref.specialfields
- jabref.bst
- jabref.exporter
- jabref.importer
- jabref.sql
- jabref.util

The metrics in GQM tree chosen for this study were extracted from the above listed packages for every release using the metrics extraction tools described in Section 3. Class level metric are translated to package level abstraction with applicable statistical measures. These metrics are visualised and analysed to answer the questions in GQM tree eventually finding the packages that are difficult to maintain. The findings are validated against the maintainability index of these packages.

A. GQM Questions

Throughout the last 10 releases of Jabref, many packages were merged or removed from the later versions. Addressing every package will be a repetition with no major insights. Therefore to restrict the size of the report, packages of the current release will be our primary focus. Also, since this study is on the root level packages of Jabref, identifying the difficult modules in the current release will inherently mean its sub-packages are also difficult to maintain.

Appropriate visualisation techniques are used to depict the findings and insights from the metrics data of the packages. Comments on the maintainability of the modules is based upon its relationship with metrics from literature. Under every question, each metric is analysed as an individual entity to interpret the maintainability of the system drawing references between each other at occasions. All the metric observations and their interpretation of each module will be combined to identify the modules difficult to maintain in the reflection points.

Q1: Which modules of Jabref are cardinal for the system?

EC is a module criticality indicator used to understand the code structure and identify critical modules in Jabref. [15] The connectivity matrix of every module in each release generated from Structure 101 was fed to UCINET to analyse and calculate the EC at Package level. NetDraw helped in analysing this data as a global dependency networks. Global dependency networks are directed graphs showing the connectivity between all modules in a network.[15] Fig 4 shows the global dependency network of Jabref version 5.

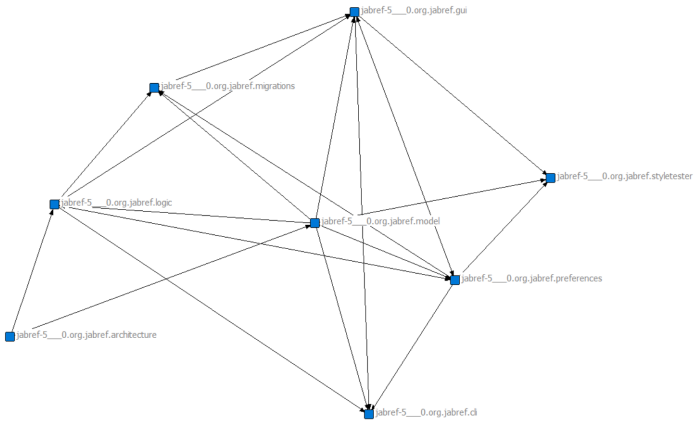

Fig 4: Global dependency network of Jabref version 5

Critical modules in each release of Jabref were identified by calibrating the global dependency network visualisation and resulting EC metric value. The global dependency networks grow complex as the number of modules increase. Therefore the global dependency networks of older versions of Jabref are complicated.

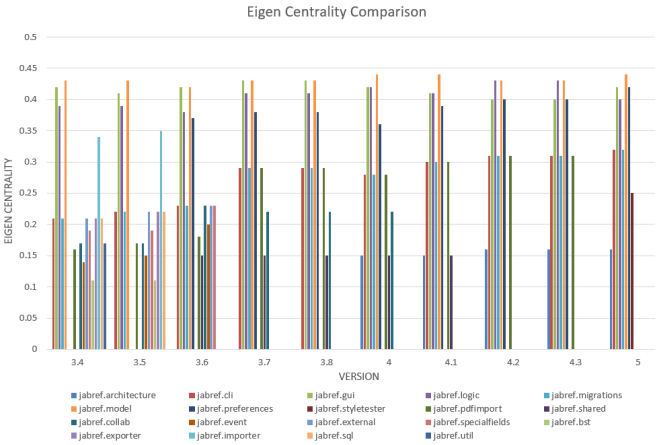

Fig 5: EC of every package across the last 10 releases of Jabref

To restrict the size of the report, a consolidated column chart representing the EC of each package in every release of Jabref is shown in Fig 5. The data used for generating this column chart is shown in Fig 6.

Eigen Centrality										
Packages	**3.4**	**3.5**	**3.6**	**3.7**	**3.8**	**4**	**4.1**	**4.2**	**4.3**	**5**
jabref.architecture						0.15	0.15	0.16	0.16	0.16
jabref.cli	0.21	0.22	0.23	0.29	0.29	0.28	0.3	0.31	0.31	0.32
jabref.gui	0.42	0.41	0.42	0.43	0.43	0.42	0.41	0.4	0.4	0.42
jabref.logic	0.39	0.39	0.38	0.41	0.41	0.42	0.41	0.43	0.43	0.4
jabref.migrations	0.21	0.22	0.23	0.29	0.29	0.28	0.3	0.31	0.31	0.32
jabref.model	0.43	0.43	0.42	0.43	0.43	0.44	0.44	0.43	0.43	0.44
jabref.preferences			0.37	0.38	0.38	0.36	0.39	0.4	0.4	0.42
jabref.styletester										0.25
jabref.pdfimport	0.16	0.17	0.18	0.29	0.29	0.28	0.3	0.31	0.31	
jabref.shared			0.15	0.15	0.15	0.15	0.15			
jabref.collab	0.17	0.17	0.23	0.22	0.22	0.22				
jabref.event	0.14	0.15	0.2							
jabref.external	0.21	0.22	0.23							
jabref.specialfields	0.19	0.19	0.23							
jabref.bst	0.11	0.11								
jabref.exporter	0.21	0.22								
jabref.importer	0.34	0.35								
jabref.sql	0.21	0.22								
jabref.util	0.17									

Fig 6: Data of EC of every package

Based on our observation and analysis, *jabref.model* consistently remained the critical module throughout the previous 10 releases followed by *jabref.gui, jabref.logic* and *jabref.preferences*. Therefore these modules would be difficult to maintain.[34][35]

Q2: Which modules of Jabref have relatively poor coupling?

Coupling is used as an indicator of code structure and complexity. Three metrics: Ca, Ce, CBO are measured to analyse the coupling in the modules of Jabref.

Ca and Ce are incoming coupling and outgoing coupling respectively.[16][19] CBO defines an inter-class dependency.[17] All the three metrics were extracted at using Metrics Reloaded plugin in IntelliJ IDEA IDE. Ca and Ce were extracted at the package level while statistical mean of CBO was considered to translate it for package level abstraction.

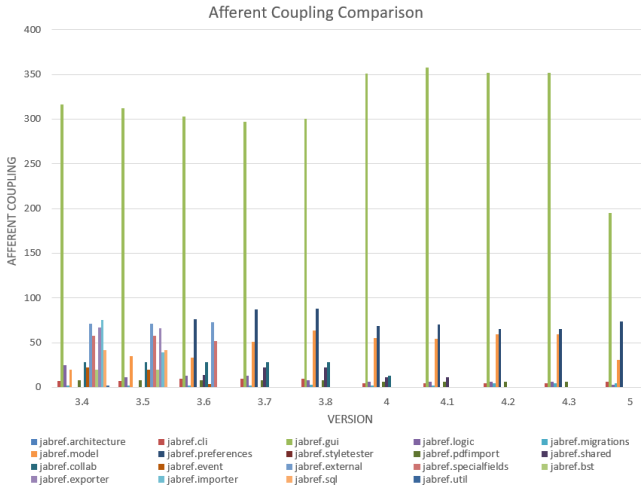

Fig 7: Ca of every package across the last 10 releases of Jabref

Fig 7 shows the column chart of Ca comparison of every package in every release of Jabref. The data used to generate this chart is shown in Fig 8.

Ca										
Packages	**3.4**	**3.5**	**3.6**	**3.7**	**3.8**	**4**	**4.1**	**4.2**	**4.3**	**5**
jabref.architecture						0	0	0	0	0
jabref.cli	7	7	10	10	10	5	5	5	5	6
jabref.gui	316	312	303	297	300	351	358	352	352	195
jabref.logic	25	11	13	13	8	6	6	6	6	3
jabref.migrations	2	2	2	2	3	2	2	5	5	5
jabref.model	20	35	33	51	64	55	54	59	59	31
jabref.preferences			76	87	88	69	70	65	65	74
jabref.styletester										0
jabref.pdfimport	8	8	8	8	8	6	6	6	6	
jabref.shared			14	22	22	11	11			
jabref.collab	28	28	28	28	28	13				
jabref.event	22	20	4							
jabref.external	71	71	73							
jabref.specialfields	58	58	52							
jabref.bst	20	20								
jabref.exporter	67	66								
jabref.importer	75	39								
jabref.sql	42	42								
jabref.util	2									

Fig 8: Data of Ca of every package across the last 10 releases of Jabref

The Ca value of module *jabref.gui* is consistently the highest across the last 10 releases followed by the *jabref.preferences, jabref.model, jabref.logic* and *jabref.cli*. Although higher Ca values are favourable for good maintainability[16][19], but it is important to analyse Ca together with Ce.

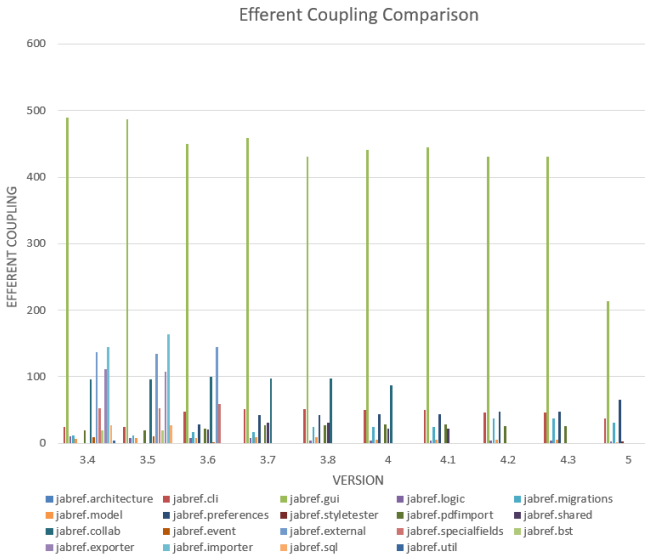

Fig 9: Ce of every package across the last 10 releases of Jabref

Fig 9 shows column chart of Ce comparison of every package in every release of Jabref. The data used to generate this chart is shown in Fig 10.

Ce										
Packages	**3.4**	**3.5**	**3.6**	**3.7**	**3.8**	**4**	**4.1**	**4.2**	**4.3**	**5**
jabref.architecture						0	0	0	0	0
jabref.cli	24	24	48	51	51	50	50	46	46	37
jabref.gui	490	487	450	459	431	441	445	431	431	213
jabref.logic	10	8	8	8	4	4	4	4	4	3
jabref.migrations	12	12	17	17	25	25	25	37	37	31
jabref.model	7	8	8	9	9	6	6	6	6	2
jabref.preferences			29	42	42	44	44	47	47	66
jabref.styletester										3
jabref.pdfimport	20	20	22	27	27	28	28	26	26	
jabref.shared			21	31	31	22	22			
jabref.collab	96	96	100	97	97	87				
jabref.event	9	10	2							
jabref.external	137	134	144							
jabref.specialfields	53	53	59							
jabref.bst	19	19								
jabref.exporter	111	108								
jabref.importer	144	164								
jabref.sql	27	27								
jabref.util	4									

Fig 10: Data of Ce of every package across the last 10 releases of Jabref

Ce value for module *jabref.gui* remains constantly higher than Ca, making it highly instable module and therefore difficult to maintain. Following *jabref.gui, jabref.preferences* has consistently high Ce along with *jabref.cli* and *jabref.migrations*. Therefore these modules are speculated to have poor maintainability.[16][19]

In contrast, Ce values of *jabref.logic* and *jabref.model* are significantly low. Lower Ce values makes them less instable

although since they have similar values or close Ca values, it is safe to term them instable modules therefore are difficult to maintain.[16][19]

There are other modules which demonstrated high Ca, Ce numbers like *jabref.external, jabref.exporter* and *jabref.importer*. But since they were merged into other packages and not a root module in the current release 5.0 of Jabref, they wont be the primary focus.

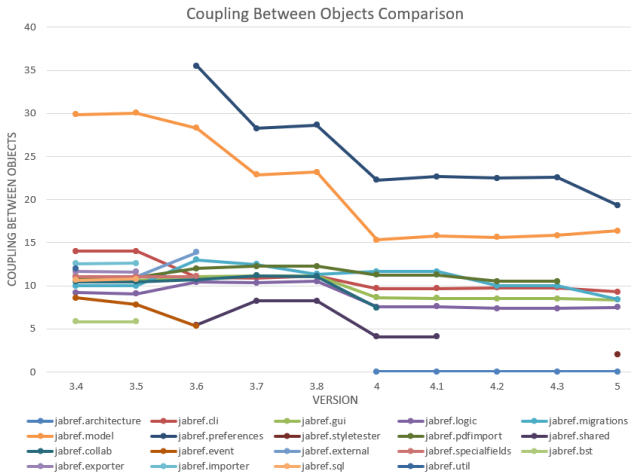

Fig 11: CBO of every package across the last 10 releases of Jabref

Fig 11 shows column chart of CBO comparison of every package in every release of Jabref. The data used to generate this chart is shown in Fig 12.

CBO										
Packages	**3.4**	**3.5**	**3.6**	**3.7**	**3.8**	**4**	**4.1**	**4.2**	**4.3**	**5**
jabref.architecture						0	0	0	0	0
jabref.cli	14	14	11	10.89	11.11	9.67	9.67	9.75	9.75	9.29
jabref.gui	10.99	10.66	11.06	11.18	11.17	8.65	8.55	8.5	8.51	8.36
jabref.logic	9.2	9.09	10.44	10.34	10.53	7.56	7.6	7.38	7.36	7.51
jabref.migrations	10	10	13	12.5	11.33	11.67	11.67	10	10	8.4
jabref.model	29.86	30.06	28.29	22.88	23.18	15.33	15.8	15.62	15.83	16.37
jabref.preferences			35.5	28.27	28.64	22.25	22.67	22.5	22.58	19.33
jabref.styletester										2
jabref.pdfimport	11	11	12	12.25	12.25	11.25	11.25	10.5	10.5	
jabref.shared			5.43	8.25	8.25	4.1	4.1			
jabref.collab	10.47	10.47	10.74	11.17	11.06	7.44				
jabref.event	8.6	7.8	5.33							
jabref.external	11.1	11.02	13.88							
jabref.specialfields	11.06	11.06	11.06							
jabref.bst	5.83	5.83								
jabref.exporter	11.67	11.59								
jabref.importer	12.62	12.63								
jabref.sql	10.63	10.79								
jabref.util	12									

Fig 12: Data of CBO of every package across the last 10 releases of Jabref

CBO values of modules *jabref.preferences, jabref.model, jabref.cli* are high across the releases. Lower CBO values are prefered for good maintainability. Therefore they can be deemed as modules difficult to maintain on basis of CBO values.[17]

Like in Ca, Ce, there are other modules with relatively high CBO value in the older releases of Jabref. These packages are merged and not the root packages in the current release 5.0 of Jabref. Therefore, to restrict the size of the report, we stick with packages in current release.

Summing up, the identified modules show high coupling indicating poor code structure creating more dependencies between each other. These dependencies directly inflict complexity overheads and performance issues. Also such codes become hard to maintain due to lower understandability caused by its high complexity. [16][17][19]

Q3: Which modules of Jabref have relatively poor cohesion?

Cohesion is used as an indicator of code structure and complexity. It is represented using LCOM. Given its supportive role as an internal attribute for defining structure and complexity along with other set of metrics, only one metric is chosen to reduce overhead. Also, LCOM is a verified metric and relevant to address cohesion.[17]

LCOM is a method-variable associativity metric used to analyse the cohesion in class.[17] LCOM was extracted at class level using Metrics reloaded plugin of IntelliJ IDE. Statistical mean of LCOM was considered to translate it for package level abstraction.

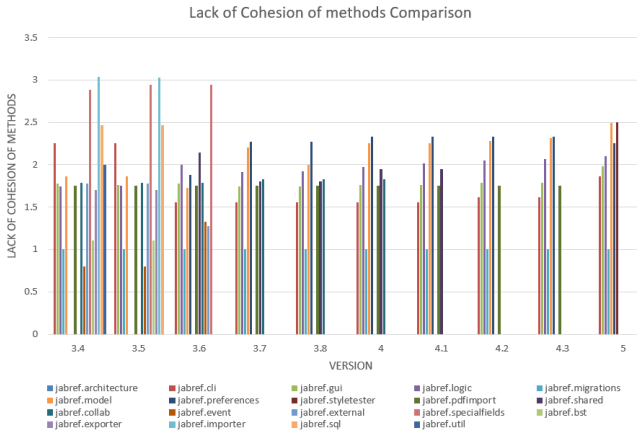

Fig 13: LCOM of every package across the last 10 releases of Jabref

Fig 13 shows column chart of LCOM comparison of every package in every release of Jabref. The data used to generate this chart is shown in Fig 14.

LCOM										
Packages	**3.4**	**3.5**	**3.6**	**3.7**	**3.8**	**4**	**4.1**	**4.2**	**4.3**	**5**
jabref.architecture						0	0	0	0	0
jabref.cli	2.25	2.25	1.56	1.56	1.56	1.56	1.56	1.62	1.62	1.86
jabref.gui	1.78	1.76	1.78	1.74	1.74	1.76	1.76	1.79	1.79	1.98
jabref.logic	1.74	1.75	2	1.91	1.92	1.97	2.02	2.05	2.07	2.1
jabref.migrations	1	1	1	1	1	1	1	1	1	1
jabref.model	1.86	1.86	1.73	2.2	2	2.25	2.25	2.28	2.31	2.49
jabref.preferences			1.88	2.27	2.27	2.33	2.33	2.33	2.33	2.25
jabref.styletester										2.5
jabref.pdfimport	1.75	1.75	1.75	1.75	1.75	1.75	1.75	1.75	1.75	
jabref.shared			2.14	1.8	1.8	1.95	1.95			
jabref.collab	1.79	1.79	1.79	1.83	1.83	1.83				
jabref.event	0.8	0.8	1.33							
jabref.external	1.78	1.78	1.28							
jabref.specialfields	2.88	2.94	2.94							
jabref.bst	1.11	1.11								
jabref.exporter	1.7	1.7								
jabref.importer	3.04	3.03								
jabref.sql	2.47	2.47								
jabref.util	2									

Fig 14: Data of LCOM of every package across the last 10 releases of Jabref

On observing the data, *jabref.preferences and jabref.model* have the high LCOM values followed by *jabref.logic* and *jabref.gui*. High LCOM value of modules reflect poor structuring of code hence making them difficult to maintain.

Other modules with high LCOM values in the previous versions of LCOM include *jabref.specialfields, jabref.collab, jabref.pdfimport, jabref.collab* and *jabref.external*. All these modules were merged in later releases.

Therefore, on conclusive note it is relevant to term the identified modules show poor code structuring with no true relationship between the methods and variables within one class thus lower its encapsulation. Lower encapsulation results in poor understanding of the modules with no reason of having member variables and member functions together. These modules start resisting changes with their unjustified cohesion. Hence, these modules will be difficult to maintain due to lack of strong cohesion.[17]

Q4: Which modules of Jabref contain complex code?
Complexity of code is represented with CC and WMC. Both the metrics were extracted using Metrics Reloaded plugin for IntelliJ IDEA IDE at class level. Statistical mean of both the metrics were considered to translate them for package level abstraction.

CC measures the linear independent paths in program flow. Complexity of code increases with increased CC.[18]

Fig 15 shows Line chart of Average CC comparison of every package in every release of Jabref. The data used to generate this chart is shown in Fig 16.

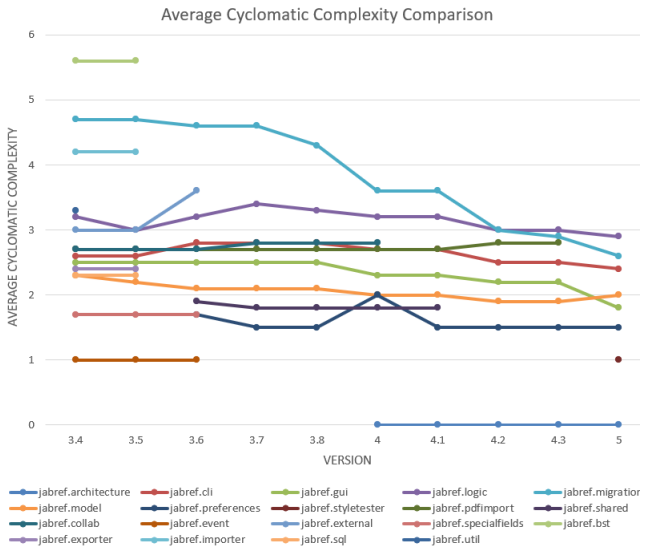

Fig 15: Average CC of every package across the last 10 releases of Jabref

As observed in the chart, average CC of the modules have decreased over the last ten releases. However, average CC of modules *jabref.migration* and *jabref.logic* remain relatively high. Since higher values of average CC increases complexity, both these modules will be difficult to maintain.[18]

Avg CC										
Packages	**3.4**	**3.5**	**3.6**	**3.7**	**3.8**	**4**	**4.1**	**4.2**	**4.3**	**5**
jabref.architecture						0	0	0	0	0
jabref.cli	2.6	2.6	2.8	2.8	2.8	2.7	2.7	2.5	2.5	2.4
jabref.gui	2.5	2.5	2.5	2.5	2.5	2.3	2.3	2.2	2.2	1.8
jabref.logic	3.2	3	3.2	3.4	3.3	3.2	3.2	3	3	2.9
jabref.migrations	4.7	4.7	4.6	4.6	4.3	3.6	3.6	3	2.9	2.6
jabref.model	2.3	2.2	2.1	2.1	2.1	2	2	1.9	1.9	2
jabref.preferences			1.7	1.5	1.5	2	1.5	1.5	1.5	1.5
jabref.styletester										1
jabref.pdfimport	2.7	2.7	2.7	2.7	2.7	2.7	2.7	2.8	2.8	
jabref.shared			1.9	1.8	1.8	1.8	1.8			
jabref.collab	2.7	2.7	2.7	2.8	2.8	2.8				
jabref.event	1	1	1							
jabref.external	3	3	3.6							
jabref.specialfields	1.7	1.7	1.7							
jabref.bst	5.6	5.6								
jabref.exporter	2.4	2.4								
jabref.importer	4.2	4.2								
jabref.sql	2.3	2.3								
jabref.util	3.3									

Fig 16: Data of Average CC of every package across the last 10 releases of Jabref

WMC is sum of CC of all local methods. Complexity of code increases with increased CC.[17]

Fig 17 shows Line chart of WMC comparison of every package in every release of Jabref. The data used to generate this chart is shown in Fig 18.

The data clearly indicates WMC value for modules *jabref.preferences* is relatively higher than other modules. *jabref.cli*, *jabref.migrations* also has a history of high WMC values although its has fallen considerably over the last few releases. Therefore, due to the high WMC values these modules will relatively be difficult to maintain. [17]

There are other modules like *jabref.importer* in the previous releases of Jabref with very high WMC values. These modules were merged in the later releases.

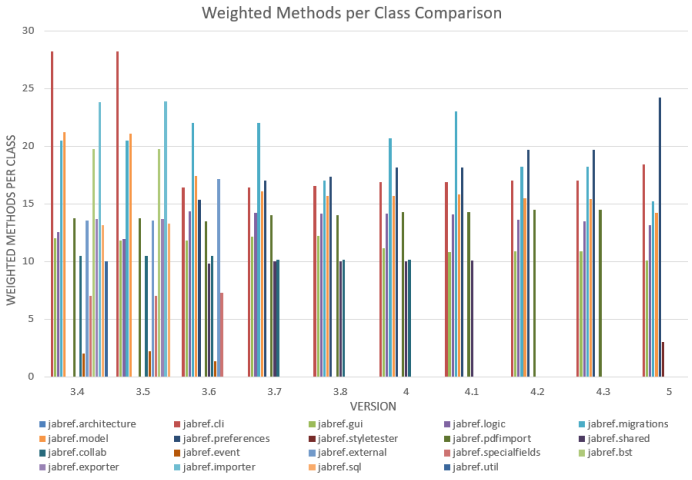

Fig 17: WMC of every package across the last 10 releases

of Jabref

WMC										
Packages	3.4	3.5	3.6	3.7	3.8	4	4.1	4.2	4.3	5
jabref.architecture						0	0	0	0	0
jabref.cli	28.25	28.25	16.44	16.44	16.56	16.89	16.89	17	17	18.43
jabref.gui	12.02	11.81	11.82	12.13	12.22	11.14	10.79	10.87	10.88	10.08
jabref.logic	12.56	11.92	14.35	14.23	14.13	14.13	14.07	13.61	13.51	13.13
jabref.migrations	20.5	20.5	22	22	17	20.67	23	18.2	18.2	15.2
jabref.model	21.2	21.08	17.43	16.09	15.7	15.67	15.79	15.48	15.39	14.23
jabref.preferences			15.38	17	17.36	18.17	18.17	19.67	19.67	24.25
jabref.styletester										3
jabref.pdfimport	13.75	13.75	13.5	14	14	14.25	14.25	14.5	14.5	
jabref.shared			9.79	10	10	10	10.05			
jabref.collab	10.47	10.47	10.47	10.17	10.17	10.17				
jabref.event	2	2.2	1.33							
jabref.external	13.56	13.54	17.16							
jabref.specialfields	7	7	7.25							
jabref.bst	19.72	19.72								
jabref.exporter	13.7	13.67								
jabref.importer	23.84	23.9								
jabref.sql	13.16	13.26								
jabref.util	10									

Fig 18: Data of WMC of every package across the last 10 releases of Jabref

The complexity of the identified packages is relatively higher creating a steep learning curve for the contributors to push code into these modules. It also influences the code understandability and demands greater efforts for revisions and restorations. Also with added complexity, performance issues arise further damaging the user experience. They slow down the releases thus holding the modules in a long cycle of development. Hence these modules will be difficult to maintain taking their high complexity into account.[17][18][19]

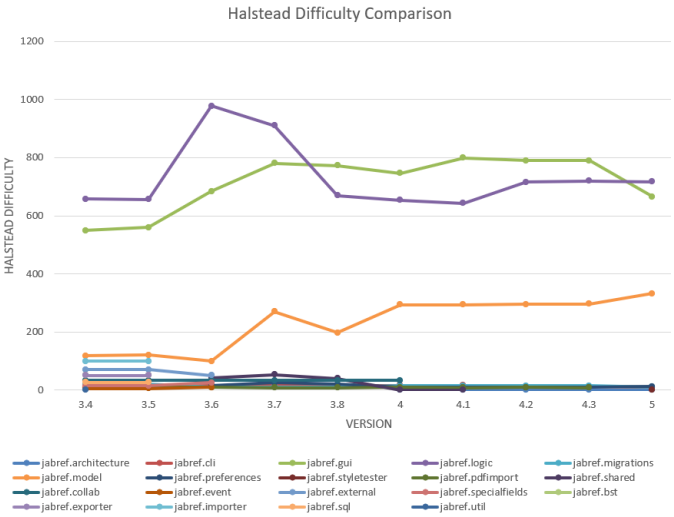

Fig 19: HD of every package across the last 10 releases of Jabref

Fig 19 shows Line chart of HD comparison of every package in every release of Jabref. The data used to generate this chart is shown in Fig 20.

Q5: What is the magnitude of readability and understandability of each Jabref module?
Understandability of the code is represented with HD, DIT, CR. HD was extracted at package level using Prest for each package in every release. DIT was extracted from Metrics Reloaded plugin of IntelliJ IDEA IDE at class level. Statistical mean of the DIT was considered to translate it for package level abstraction. CR was extracted using a combination of tools. First, TLOC was extracted from Metrics reloaded of IntelliJ IDEA IDE followed by NCLOC extraction from python CLI metrics tool Lizard. Both these metrics were extracted at package level. Then, the difference of TLOC and NCLOC was recorded to obtain the Commented Lines of Code (CLOC) for each module in every release. Finally, ratio of CLOC to TLOC was computed to derive the CR of each package.

HD										
Packages	3.4	3.5	3.6	3.7	3.8	4	4.1	4.2	4.3	5
jabref.architecture						0	0	0	0	0
jabref.cli	12.5	12.5	16.67	16.67	16.67	16.67	16.67	12.5	12.5	9.9
jabref.gui	549.7	560.21	684.04	780.35	773.08	746.13	799.22	789.32	789.32	665.9
jabref.logic	657.75	656.14	978.11	910.49	669.56	654.22	643.05	715.9	720.48	716.62
jabref.migrations	20	20	11.11	11.11	11.11	14.29	14.29	14.29	14.29	11.11
jabref.model	117.29	120.54	99.78	269.82	197.06	294.31	294.31	294.95	296.06	332.25
jabref.preferences			16.67	25	20	10	10	10	10	12.5
jabref.styletester										0
jabref.pdfimport	10	10	10	9.09	9.09	9.9	9.09	9.9	9.09	
jabref.shared			41.67	53.44	40.23	0	0			
jabref.collab	33.33	33.33	33.33	33.33	33.33	33.33				
jabref.event	5.56	5.56	9.09							
jabref.external	70	70	50							
jabref.specialfields	14.29	14.29	25							
jabref.bst	50	50								
jabref.exporter	50	50								
jabref.importer	99.99	99.99								
jabref.sql	26.72	26.72								
jabref.util	1.49									

Fig 20: Data of HD of every package across the last 10 releases of Jabref

The HD of *jabref.logic* and *jabref.gui* is consistently high across the releases. Also HD of *jabref.model* has increased considerably. Since lower values of HD are supported for better maintainability, these modules will be hard to maintain.[18][20]

HD reflects upon the number of operands and operators in source code. Higher number of either makes the code complex to understand.[18][20]

The DIT is an ancestral node count of a class. Deeper level of inheritance drops the readability and understandability of code.[17]

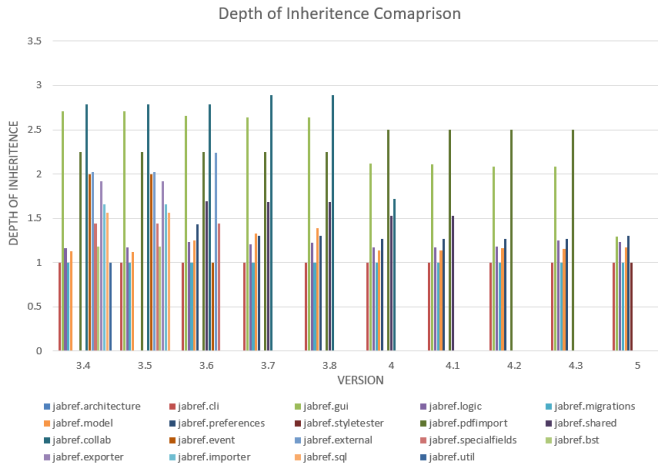

Fig 21: DIT of every package across the last 10 releases of Jabref

Fig 21 shows Column chart of DIT comparison of every package in every release of Jabref. The data used to generate this chart is shown in Fig 22.

From the data, The DIT of all modules has fallen significantly in the latest release of Jabref. Although, modules *jabref.preferences* and *jabref.gui* have history of many ancestors. Also having a high HD contributes to the fact that these modules will be difficult to maintain due to its complexity of understanding the code.

DIT										
Packages	3.4	3.5	3.6	3.7	3.8	4	4.1	4.2	4.3	5
jabref.architecture						0	0	0	0	0
jabref.cli	1	1	1	1	1	1	1	1	1	1
jabref.gui	2.71	2.71	2.66	2.64	2.64	2.12	2.11	2.08	2.08	1.29
jabref.logic	1.16	1.17	1.23	1.21	1.22	1.17	1.17	1.18	1.25	1.23
jabref.migrations	1	1	1	1	1	1	1	1	1	1
jabref.model	1.13	1.12	1.25	1.33	1.39	1.14	1.14	1.16	1.15	1.17
jabref.preferences			1.43	1.3	1.3	1.27	1.27	1.27	1.27	1.3
jabref.styletester										1
jabref.pdfimport	2.25	2.25	2.25	2.25	2.25	2.5	2.5	2.5	2.5	
jabref.shared			1.69	1.68	1.68	1.53	1.53			
jabref.collab	2.79	2.79	2.79	2.89	2.89	1.72				
jabref.event	2	2	1							
jabref.external	2.02	2.02	2.24							
jabref.specialfields	1.44	1.44	1.44							
jabref.bst	1.18	1.18								
jabref.exporter	1.92	1.92								
jabref.importer	1.66	1.66								
jabref.sql	1.56	1.56								
jabref.util	1									

Fig 22: Data of DIT of every package across the last 10 releases of Jabref

CR defines the density of comments in code assisting code readability and understandability. Higher CR is always appreciated given the comments precisely explain the corresponding code. [18]

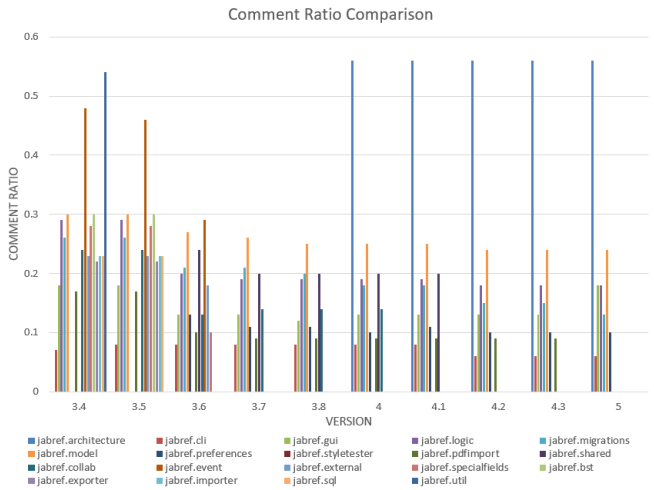

Fig 23: CR of every package across the last 10 releases of Jabref

Fig 23 shows Column chart of CR comparison of every package in every release of Jabref. The data used to generate this chart is shown in Fig 24.

CR										
Packages	3.4	3.5	3.6	3.7	3.8	4	4.1	4.2	4.3	5
jabref.architecture						0.56	0.56	0.56	0.56	0.56
jabref.cli	0.07	0.08	0.08	0.08	0.08	0.08	0.08	0.06	0.06	0.06
jabref.gui	0.18	0.18	0.13	0.13	0.12	0.13	0.13	0.13	0.13	0.18
jabref.logic	0.29	0.29	0.2	0.19	0.19	0.19	0.19	0.18	0.18	0.18
jabref.migrations	0.26	0.26	0.21	0.21	0.2	0.18	0.18	0.15	0.15	0.13
jabref.model	0.3	0.3	0.27	0.26	0.25	0.25	0.25	0.24	0.24	0.24
jabref.preferences			0.13	0.11	0.11	0.1	0.11	0.1	0.1	0.1
jabref.styletester										0
jabref.pdfimport	0.17	0.17	0.1	0.09	0.09	0.09	0.09	0.09	0.09	
jabref.shared			0.24	0.2	0.2	0.2	0.2			
jabref.collab	0.24	0.24	0.13	0.14	0.14	0.14				
jabref.event	0.48	0.46	0.29							
jabref.external	0.23	0.23	0.18							
jabref.specialfields	0.28	0.28	0.1							
jabref.bst	0.3	0.3								
jabref.exporter	0.22	0.22								
jabref.importer	0.23	0.23								
jabref.sql	0.23	0.23								
jabref.util	0.54									

Fig 24: Data of CR of every package across the last 10 releases of Jabref

The CR of modules *jabref.model, jabref.logic* and *jabref.gui* are relatively higher across the releases of jabref. Since all three are critical modules with huge code size, comments are crucial to brief about the code lines. However, *jabref.styletester, jabref.cli* and *jabref.preferences* have abysmal CR. Therefore, the later modules will have poor maintainability according to CR.

It is important to note that the comment ratio of *jabref.architecture* is higher than all other modules. But, given its code size of 9 TLOC and 0 complexity, the module is maintainable by default. Hence, we are targeting the critical modules primarily to discover their maintainability.

Based on the observations from individual metrics, modules *jabref.preferences* and *jabref.cli* have poor understandability given its poor CR. Although *jabref.gui, jabref.logic* and *jabref.model* having higher HD are better compensated with good CR. Also with high DIT, *jabref.preferences* struggle

with higher dependencies. This makes the module tough to contribute upon.[17][18][20]

Q6: Which modules of Jabref witnessed an increased and frequent change in code size across the releases?
Code size is represented using NCLOC and NOM. Both the metrics were extracted at package level with Python CLI code analyser Lizard. Code size is inversely proportional to maintainability. With added functionality there always is a need for greater resources and efforts. Thus as the code size increases, complexity increases and understandability decreases eventually making the module difficult to maintain. Hence smaller code sizes are preferred. [18][19]

NCLOC are source code without comments and contributes majorly to the size of the code. High NCLOC means lower maintainability due to increased size. [18][19]

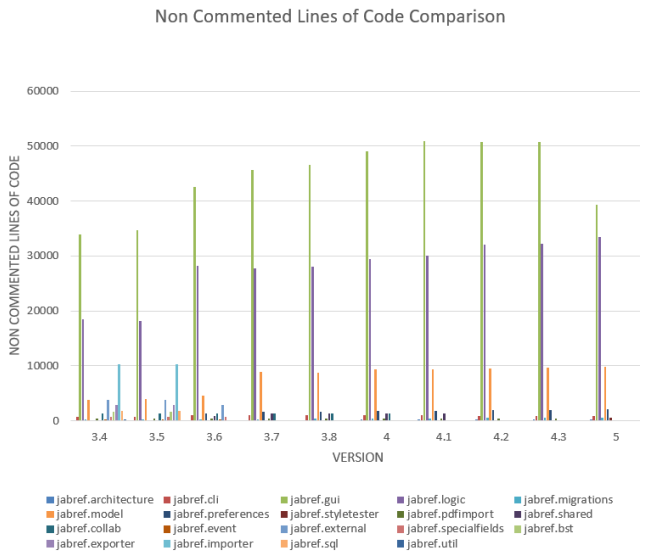

Fig 25: NCLOC of every package across the last 10 releases of Jabref

Fig 25 shows Column chart of NCLOC comparison of every package in every release of Jabref. The data used to generate this chart is shown in Fig 26.

NCLOC										
Packages	**3.4**	**3.5**	**3.6**	**3.7**	**3.8**	**4**	**4.1**	**4.2**	**4.3**	**5**
jabref.architecture						4	4	4	4	4
jabref.cli	666	666	966	975	976	987	987	894	894	836
jabref.gui	33,955.00	34,725.00	42,507.00	45,690.00	46,556.00	49,106.00	50,954.00	50,692.00	50,771.00	39,309.00
jabref.logic	18,466.00	18,231.00	28,151.00	27,717.00	27,991.00	29,489.00	30,118.00	32,125.00	32,191.00	33,444.00
jabref.migrations	241	241	272	271	330	401	434	550	566	493
jabref.model	3,823.00	3,898.00	4,485.00	8,884.00	8,786.00	9,321.00	9,286.00	9,569.00	9,583.00	9,796.00
jabref.preferences			1,245.00	1,556.00	1,564.00	1,751.00	1,755.00	1,880.00	1,880.00	2,110.00
jabref.styletester										559
jabref.pdfimport	403	403	407	431	431	435	435	431	431	
jabref.shared			922	1,322.00	1,323.00	1,324.00	1,342.00			
jabref.collab	1,360.00	1,360.00	1,371.00	1,286.00	1,278.00	1,271.00				
jabref.event	59	63	29							
jabref.external	3,742.00	3,737.00	2,862.00							
jabref.specialfields	697	699	696							
jabref.bst	1,640.00	1,640.00								
jabref.exporter	2,872.00	2,865.00								
jabref.importer	10,235.00	10,266.00								
jabref.sql	1,713.00	1,722.00								
jabref.util	59									

Fig 26: Data of NCLOC of every package across the last 10 releases of Jabref

As expected, the critical modules *jabref.gui, jabref.logic* and *jabref.model* have huge NCLOC. Also *jabref.preferences* follows *jabref.model* in NCLOC ranking. Therefore due to increased code size these packages will be difficult to maintain.[18][19]

NOM counts the number of functions within each package. Additional functionality creates complexity and understandability overhead alongside increased code and hence is preferred to be kept low for good maintainability.[21] However, this is impossible considering the scale and tasks Jabref demands.

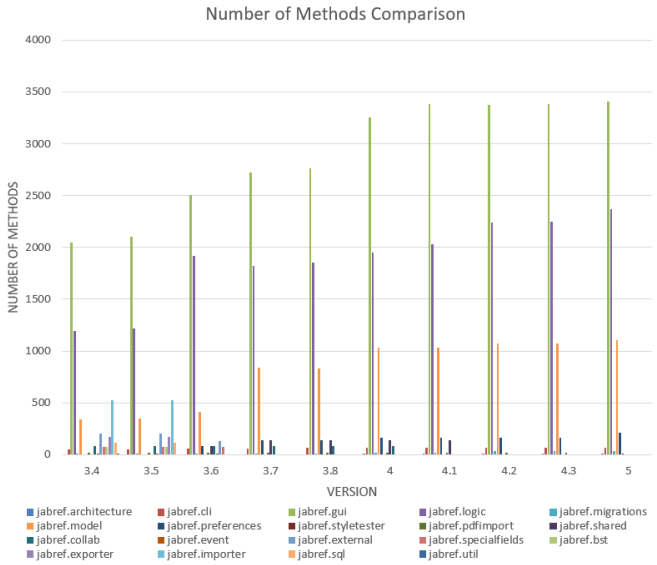

Fig 27: NOM of every package across the last 10 releases of Jabref

Fig 27 shows Column chart of NOM comparison of every package in every release of Jabref. The data used to generate this chart is shown in Fig 28.

NOM										
Packages	**3.4**	**3.5**	**3.6**	**3.7**	**3.8**	**4**	**4.1**	**4.2**	**4.3**	**5**
jabref.architecture						1	1	1	1	1
jabref.cli	53	53	63	63	64	67	67	67	67	65
jabref.gui	2049	2104	2507	2725	2762	3252	3381	3377	3379	3406
jabref.logic	1195	1217	1921	1825	1852	1953	2030	2239	2246	2366
jabref.migrations	10	10	11	11	14	20	22	34	35	32
jabref.model	341	351	417	837	832	1032	1033	1072	1073	1106
jabref.preferences			83	141	144	161	161	168	168	210
jabref.styletester										6
jabref.pdfimport	21	21	21	22	22	22	22	22	22	
jabref.shared			85	138	138	138	138			
jabref.collab	87	87	87	80	80	80				
jabref.event	10	11	4							
jabref.external	207	207	131							
jabref.specialfields	73	73	75							
jabref.bst	79	79								
jabref.exporter	173	173								
jabref.importer	529	529								
jabref.sql	114	115								
jabref.util	3									

Fig 28: Data of NOM of every package across the last 10 releases of Jabref

Following the same trend as NCLOC, the NOM of all the three critical modules *jabref.gui, jabref.logic* and *jabref.model* are extremely high along with *jabref.preferences*. Therefore, the maintainability of these modules will be extremely difficult.

Also there are other modules in previous versions of Jabref with high NOM but were merged in later releases.

Code size directly impacts the complexity and understandability of code. Huge code sizes demand greater resources and effort to understand the code. Also with heavy functionality, complexity of the code spikes high with numerous dependencies. These functionalities also demand tasks from other modules therefore creating additional coupling. Overall, the identified modules as anticipated have poor coupling, understandability and high complexity due to their enormous code size as result of critical functionalities they address. Therefore these modules will be difficult to maintain considering their code size. [18][19][21]

B. Reflection Points

A: Which of the product modules will be more difficult to maintain? And Why?
The internal attributes identified to interpret the maintainability of the modules bind and compliment the findings of each other. The modules with poor code structure as a result of their poor cohesion and high coupling, high complexity influenced by its structure and functionality, low understandability as a product of high complexity and huge code size as the root for all other attributes, are more difficult to push code to or process revision and restorations.

From the analysis, *jabref.preferences, jabref.gui, jabref.logic* and *jabref.model* in the current release of Jabref will be difficult to maintain considering their poor code structure, high complexity, low understandability and massive code size. Given the nature of being critical modules for the system, *jabref.gui, jabref.logic* and *jabref.model* naturally tend to demand higher resources and greater efforts contributing to them. Although every possible attempt is made to make them maintainable with good CR, and optimal dependencies. *jabref.preferences* struggles in every sphere of the internal attributes with undesired results from chosen metrics. Despite its relatively smaller size, *jabref.preferences* catapults huge complexity and poor understanding alongside its bad code structure. The other modules in the current release *jabref.cli* and *jabref.migrations* show similar characteristics of *jabref.preferences*.

Ironically, *jabref.preferences, jabref.cli* and *jabref.migrations* are more difficult to maintain then the critical modules considering their limited role yet amount poor structure and complexity they represent is unjustified. Critical modules are inherently difficult to maintain due to their diverse functionality and demanding tasks. Also with majority of modules in the previous versions were merged into the critical modules further creating an extra overhead. Yet the modules are well packed to support revisions and restorations. Therefore the *jabref.preferences* will be the most difficult to maintain module followed by *jabref.migrations, jabref.cli, jabref.logic, jabref.gui* and finally *jabref.model*. It is obvious and clear looking at the ranking of the modules more difficult to maintain then other have huge complexity and poor structuring over low understandability. Code size also contributes to maintainability but with right measures codes of huge sizes can also be simplified for maintenance.

Our results correspond with the maintainability index of each package as shown in Fig 29.

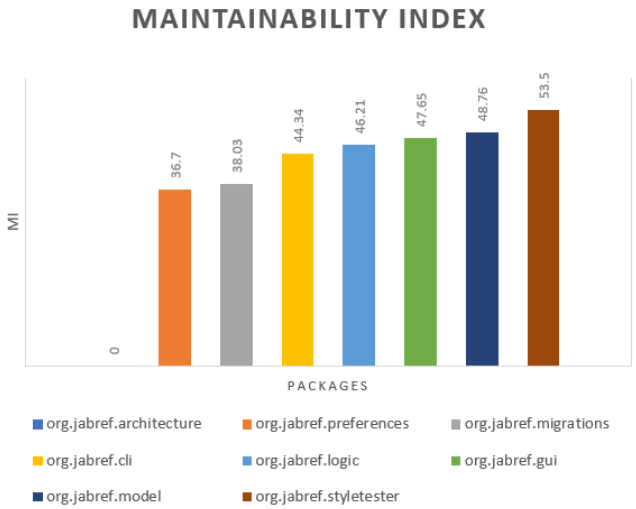

Fig 29: Maintainability Index of packages in current release of Jabref.

Maintainability Index(MI) represents the ease of maintenance efforts for a given module. Higher value of MI assures the packages to be more maintainable.[21] Our results replicate the findings from MI perfectly.

There are no comments made on module *jabref.architecture* since its a configuration package for choosing core java libraries housing only 4 NCLOC and zero complexity with no changes or updates post its initial commit in version 4.0. Hence the maintainability index is also shown as 0 which is imaginary and not relevant.

B: What can be done to improve the maintainability of these modules ?
Maintainability is primarily hit with poor code structuring and high complexity. Low coupling and high cohesion contribute to good code structure indirectly improving understandability. The inter-module dependencies must be optimised and appropriate objects must be packed in relevant classes to improve cohesion thus reducing the outgoing object calls. With controlled dependencies understandability improves implicitly. Precise and meaningful comments must be added to the source code to improve the understandability therefore reducing the overall effort overhead. Complexity is a key factor to address to improve maintainability. Given most

of the modules with low maintainability house complex code, the branching conditions and program flow must be controlled to optimal level for better understandability with reduced complexity. Functions should be broken down to address niche and particular task to minimise the monarchy of dependency and improve class cohesion. This approach also minimised per function cyclomatic complexity thus making it easier to maintain the system in parts. Size is an uncontrollable factor. With the requirement and increase in functionality, more lines of code and functions must be contributed. Although, optimised variants of syntax can be utilised to minimise the code size. Also it is important to avoid unwanted and inefficient algorithm designs adding complexity with no specific functionality or advantage to the system.

C: How various internal product attributes(e.g., cohesion, coupling and complexity of modules) are related to module size. Do modules with large size have poor structure as well?
Code size positively correlate with most of the internal product attributes. With increase in code size, number of functions and complexity of the code increases thus lowering the overall understanding. Code structure remains to be non-trivial with code size. Cohesion and coupling are prospects of code design and data flow, therefore code size cannot be justified as a valid component to address the change in coupling and cohesion values used to acknowledge the overall code structure of the system. If the code design is optimised for minimal dependencies and high intra-module reliance, then code size may contribute positively for code structure else not.

This is valid in the current outlook of Jabref. The modules *jabref.preferences, jabref.gui, jabref.logic* and *jabref.model* showcase poor code structure with huge code size. This can be accounted as a result of inefficient design approach to optimise the dependencies and increase cohesion. Therefore there is a clear scope of improvement in code structure for these modules to increase their maintainability.

D: Since you are studying last ten releases of the product, you can identify the modules that were being updated more frequently and substantially. Are these the ones that have poor code structure? Do the newly added modules possess poor/better code structure?
The metrics extracted from last ten releases of Jabref were analysed specifically in terms of Code size to examine the frequency of updates on each package. NCLOC and NOM were primarily used to acknowledge the results. Table 4 shows the frequency of change for each package. It is evident that package with high change frequency have poor code structuring. But, as discussed in the previous reflection point it is not completely relevant to address code structuring in terms of code size alone. Considering the change log of Jabref available at each release, it is clear that multiple functionalities have been added overtime. Code size is a result of these additions. On contrary it is also possible that alongside these additions there were some structural changes involved to improve the code structure. Therefore, although the modules with huge code size represent poor code structure, the cause behind maybe non-trivial.

The code structure of newly added modules is comparatively better than the legacy modules. This also comes with their importance and criticality in the system. Given their supporting roles they only address a specific part of the functionality thus optimising their dependencies and sticking with objects relevant for their classes.

Table 4: Frequency of change for each package

Package	**Frequency of Updation and Revision**
jabref.architecture	None
jabref.cli	Moderate
jabref.gui	High
jabref.logic	High
jabref.migrations	Moderate
jabref.model	High
jabref.preferences	High
jabref.styletester	Low
jabref.pdfimport	Low
jabref.shared	Moderate
jabref.collab	Moderate
jabref.event	Low
jabref.external	Low
jabref.specialfields	Low
jabref.bst	None
jabref.exporter	Low
jabref.importer	Low
jabref.sql	Low
jabref.util	None

E: Do the critical modules with huge code size always be difficult to understand and maintain?
EC points out the critical modules in the dependency network of the system. Critical modules are notoriously difficult to maintain given the functionality they ship. They house huge code size in comparison with other modules. Code size contributes to increased complexity. When complexity increases, the effort to modify also increases. But, this doesn't always translate to poor understandability. Increased code size sometimes also witness an improvement in comment ratio thus making the module more understandable. Taking the example of *jabref.model*, despite its enormous code size, the understandability of this module is relatively higher in comparison with other critical modules as a result of increased comment density. Therefore it is possible that critical modules are contributed to with maintainability into primary consideration. Hence, critical modules with huge code size are not always difficult to understand and maintain.

IX. Discussions

A. Related Work

Dallal et al. [31] reported maintainability of Jabref classes using internal attributes. The primary objective of the study was to empirically analyse the relationship between internal attributes and maintainability of Object Oriented systems. The maintainability of classes of Jabref and 2 other open source system were recorded with two maintainability indicators: Number of revised lines of code and Number of revisions involving the class. The impacts of internal attributes were evaluated using statistically based prediction models. The classes with higher cohesion values and lower code size and coupling demonstrated better maintainability. Our study further confirms these findings at package level of Jabref. Since packages are composition of classes, it is inherently evident to correspond with the behaviours of class. Hence, packages with poor code structure accounting to its poor cohesion and high coupling alongside huge code size are difficult to maintain. Also, we normalised the packages with their code size thus adding another dimension to perspective on reporting maintainability.

Ulan et al. [32] analysed the maintainability of Jabref using CK metric Suite. The results showcased classes with higher CK metric values struggled with poor maintainability. Our study corresponds to these result with packages having poor CK metric values define extensive resources and efforts for making modifications or restorations to the code. In comparison, we have evaluated our study at package level abstraction with metrics extending beyond the scope of CK metrics suite to capture a wider perspective. Also, with goal question metrics based empirical study, our study grounds the results in data with significant scientific and statistical backing.

Tobias et al. [33] reported the impacts of refactoring Jabref architecture in 2015 using code churn and code ownership. The refactoring restricted model package from importing objects associated to gui and logic. Also, logic was cut out of imports from gui. This change resulted in translation of files with violations to normal code churn thus improving their maintainability. This also corresponds to our results, making model and logic more stable and maintainable in comparison to gui package.

B. Reflections on the Project

The study helped us in understanding the practical way to approach a project, to analyse the various aspects of any code and draw inferences which would define the quality of the code helping in building better software. In this project, we learnt about the open-source software Jabref used to manage bibliographic aspects of research. As to fulfil the aim i.e., to identify the packages which pose difficulty in maintenance, we studied the related work done on Jabref thus improving our reach and limits of metric knowledge by exposing us to different measures. By using the GQM paradigm, we constructed the goals, questions and thus selected relevant metrics. This process helped us in understanding on how to methodically break the problem and approach it. We have also gained first hand experience on construction of a GQM tree. In specific, knowledge on the internal attributes and their correlation with software maintainability was attained. To extract the metric data, research and trials had been done on various tools which aided in a large way. This process boosted our skills and knowledge on the available, in-trend tools which are used currently for both academic and industrial code analysis. Then, with data collection, we learned to manage huge chunks of data with efficient schema aiding clean and insightful visualizations. We also learnt selecting scale types, deriving inferences and justifying the inferred results through applicable statistical methods. Throughout the course of the project, we understood the importance of team and collaboration in building meaning outputs.

X. Threats to Validity

Internal Validity: The main aim of the study was to identify the packages that need greater maintenance and the goals and hence the questions were framed accordingly. The metrics that were thus identified, were chosen based on their relevance to answer the questions and there could be other metrics to answer the same questions too. The justifications for the chosen metrics has been provided in this document in the earlier sections.

External Validity: The study discusses the analysis on maintainability of Jabref by the observations of the few relevant metrics considered. Generalization of the results, hence, may not be done as the results obtained for a different set of metrics could be varied. Also the tools that were used have provided with certain values which could be different on usage of other tools.

Construct Validity: The metric data collected in this study has been selected based on insights from previous research and our questions were formulated precisely to meet the goals of our study. The metrics taken for this study satisfy the purpose of the research and in getting the desired result.

Conclusion Validity: The metrics defined also are significant in understanding the maintainability of the packages in the software and the co-relation between the metrics and maintainability was derived from studying the research in the subject and through statistical inference.

XI. Conclusion

The report generated summarizes the maintainability study made on Jabref, an open-source software developed to help cross-platform citation and management of references. The objective of this study was to identify the packages of Jabref which pose difficulty for maintenance. To achieve this, a Goal

Question Metric(GQM) based empirical study was proposed. A GQM tree specific to this study was constructed using the identified internal attributes and their metrics relevant to address the goal. The metrics were extracted from last 10 releases of Jabref using metric extraction tools. The complete report was conducted as a Case Study. With appropriate visualisations and statistical measures, each packages of every release were analysed using correlation between metric values and maintainability as reported in previous studies. The interpretations of every metric were summarised to answer their respective question in the GQM tree. The current release of Jabref version 5.0 is the primary focus of this study since majority of older packages are merged into existing packages.

From our analysis, *jabref.preferences, jabref.gui, jabref.logic* and *jabref.model* in the current release of Jabref will be difficult to maintain considering their poor code structure, high complexity, low understandability and massive code size followed by *jabref.cli* and *jabref.migrations*. There are no comments made on *jabref.architecture* given its a configuration package for choosing core java libraries housing only 4 NCLOC and zero complexity with no changes or updates post its initial commit in version 4.0. Our results compliment with maintainability index of each module of release 5.0. Thus verifying the validity of the study.

REFERENCES

[1] Van Solingen, Rini, and Egon Berghout. The Goal/Question/Metric Method: a practical guide for quality improvement of software development. McGraw-Hill, 1999.

[2] Rini Van Solingen, Vic Basili, Gianluigi Caldiera, and H Dieter Rombach. Goal question metric (gqm) approach. Encyclopedia of software engineering, 2002.

[3] Zainal, Zaidah. "Case study as a research method." Jurnal Kemanusiaan 5, no. 1 (2007).

[4] Schell, Charles. "The value of the case study as a research strategy." Manchester Business School 2 (1992): 1-15.

[5] Bansal, M., Agrawal, C.P., 2014. Critical Analysis of Object Oriented Metrics in Software Development, in: 2014 Fourth International Conference on Advanced Com- puting Communication Technologies. IEEE, pp. 197–201. doi:10.1109/ACCT.2014.106.

[6] Sanjay Kumar Dubey and Ajay Rana. Assessment of maintainability metrics for object-oriented software sys- tem. ACM SIGSOFT Software Engineering Notes, 36(5):1– 7, 2011.

[7] De Silva, D.I., Weerawarna, N., Kuruppu, K., Ellepola, N., Kodagoda, N., 2013. Applicability of three cognitive complexity metrics, in: 2013 8th International Conference on Computer Science Education. IEEE,

[8] Zhang, Wei, Liguo Huang, Vincent Ng, and Jidong Ge. "SMPLearner: learning to predict software maintainability." Automated Software Engineering 22, no. 1 (2015): 111-141. pp. 573–578. doi:10.1109/ICCSE.2013.6553975.

[9] Riaz, Mehwish, Emilia Mendes, and Ewan Tempero. "A systematic review of software maintainability prediction and metrics." In Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement, pp. 367-377. IEEE Computer Society, 2009.

[10] Heitlager, Ilja, Tobias Kuipers, and Joost Visser. "A practical model for measuring maintainability." In Quality of Information and Communications Technology,2007.QUATIC2007.6th International Conference on the, pp. 30-39. IEEE, 2007.

[11] de AG Saraiva, Juliana, Micael S. De França, Sérgio CB Soares, J. C. L. Fernando Filho, and Renata MCR de Souza. "Classifying metrics for assessing object-oriented software maintainability: A family of metrics' catalogs." Journal of Systems and Software 103 (2015): 85-101.

[12] Barbara Kitchenham. Procedures for performing systematic reviews. Keele, UK, Keele University, 33(2004):1–26, 2004

[13] Zainal, Zaidah. "Case study as a research method."Jurnal Kemanusiaan 5, no. 1 (2007).

[14] Schell, Charles. "The value of the case study as a research strategy." Manchester Business School 2 (1992): 1-15.

[15] Nguyen, T.H.D., Adams, B., Hassan, A.E., 2010. Studying the impact of dependency network measures on software quality, in: 2010 IEEE International Conference on Software Maintenance. IEEE, pp. 1–10. doi:10.1109/ICSM.2010.5609560.

[16] Almugrin, Saleh Albattah, Waleed Melton, Austin. (2016). Using Indirect Coupling Metrics to Predict Package Maintainability and Testability. Journal of Systems and Software. 121. 10.1016/j.jss.2016.02.024.

[17] Dubey, S.K., Rana, A., 2011. Assessment of maintainability metrics for object-oriented software system. ACM SIGSOFT Softw. Eng. Notes 36, pp. 1–7. doi:10.1145/2,020,976.2020983.

[18] Frappier, Marc, Stan Matwin, and Ali Mili. "Software metrics for predicting maintainability." Software Metrics Study: Tech. Memo 2 (1994).

[19] Hegedűs P., Bakota T., Illés L., Ladányi G., Ferenc R., Gyimóthy T. (2011) Source Code Metrics and Maintainability: A Case Study. In: Kim T. et al. (eds) Software Engineering, Business Continuity, and Education. ASEA 2011. Communications in Computer and Information Science, vol 257. Springer, Berlin, Heidelberg

[20] D. Coleman, D. Ash, B. Lowther and P. Oman, "Using metrics to evaluate software system maintainability," in Computer, vol. 27, no. 8, pp. 44-49, Aug. 1994, doi: 10.1109/2.303623.

[21] Dagpinar, M. Weber, Jens. (2003). Predicting Maintainability with Object-Oriented Metrics - An Empirical Comparison. Reverse Engineering - Working Conference Proceedings. 155- 164. 10.1109/WCRE.2003.1287246.

[22] N. Fenton and J. Bieman, Software Metrics: A Rigorous and Practical Approach, Third Edition. CRC Press, 2014.

[23] Simon, Martin, Linus W. Dietz, Tobias Diez and Oliver Kopp. "Analyzing the Importance of JabRef Features from the User Perspective." ZEUS (2019).

[24] Metrics Reloaded - https://blog.jetbrains.com/idea/2014/09/touring-plugins-issue-1/ accessed on 22/05/2020

[25] Lizard - https://libraries.io/pypi/lizard accessed on 22/05/2020

[26] Kocaguneli, Ekrem Tosun, Ayse Bener, Ayse Turhan, Burak Caglayan, Bora. (2009). Prest: An Intelligent Software Metrics Extraction, Analysis and Defect Prediction Tool.. 637-642.

[27] JHawk - http://j-hawk.sourceforge.net/ accessed on 22/05/2020

[28] Borgatti, S.P., Everett, M.G. and Freeman, L.C. 2002. Ucinet for Windows: Software for Social Network Analysis. Harvard, MA: Analytic Technologies.

[29] Structure 101 - https://structure101.com/ accessed on 22/05/2020

[30] Borgatti, S.P., 2002. NetDraw Software for Network Visualization. Analytic Technologies: Lexington, KY

[31] Al Dallal, Jehad. (2013). Object-oriented class maintainability prediction using internal quality attributes. Information and Software Technology. 55. 2028–2048. 10.1016/j.infsof.2013.07.005.

[32] Ulan, Maria, Welf Löwe, Morgan Ericsson, and Anna Wingkvist. "Towards meaningful software metrics aggregation." In Proceedings of the 18th Belgium-Netherlands Software Evolution Workshop. 2019.

[33] Olsson, Tobias, Morgan Ericsson, and Anna Wingkvist. 2017. "The Relationship of Code Churn and Architectural Violations in the Open Source Software JabRef."ACM. doi:10.1145/3129790.3129810.

[34] Chun Yong Chong, Sai Peck Lee,Analyzing maintainability and reliability of object-oriented software using weighted complex network, Journal of Systems and Software,Volume 110,2015,Pages 28-53,ISSN 0164-1212,https://doi.org/10.1016/j.jss.2015.08.014.

[35] A. Shahjahan, W. haider Butt and A. Z. Ahmad, "Impact of refactoring on code quality by using graph theory: An empirical evaluation," 2015 SAI Intelligent Systems Conference (IntelliSys), London, 2015, pp. 595-600, doi: 10.1109/IntelliSys.2015.7361201.