JOnion: Code Metrics Calculation

Code Quality Detection Tool

Feng Gao
Lassonde School of Engineering
York University
Toronto, Canada
gaofeng@cse.yorku.ca

Emad Gohari
Lassonde School of Engineering
York University
Toronto, Canada
gohari@cse.yorku.ca

Abstract

Typically, research on defect-proneness defines metrics to capture the complexity of software and builds models that relate these metrics to defect-proneness. Following the trend of code quality study, a demand on an efficient and accurate code complexity metrics calculation tool currently is raised urgently. As the code complexity researchers, our ultimate goal for this project is to create an integrated tool called JOnion able to calculate different code complexity metrics based on the our prototype of assignment parser created during the period of the course assignment. In this tool's development cycle, we enhance the program to provide the ability of more software metrics calculation with better precision. After the development, JOnion is tested on our sufficient designated unit test cases to assure about its validity. Eventually, the tool is evaluated against some other software calculation tools available on the web. For the evaluation phase, dataset containing three representative open source software projects will be used. We try to argue the difference between other tools and JOnion with reasonable explanations. At last, we report our findings with providing detailed information about works in each step and presenting numerical results.

Keywords

Code Metrics; Code Complexity; Metrics calculation; AST trees; Static Analysis; Code Quality; Java Parser;

I. Introduction

In Software Engineering similar with any other science discipline, a high accuracy in the measurements is needed. In terms of a relatively high quality software product, it is necessary to measure the quality of developed software during the development cycle. Good software metrics give developers and researchers a thorough comprehension about the quality of the code developed or even the projects studied on. Once the metrics generated, we are able to monitor the code health condition during development cycle and prevent potential failures in order to ensure a high quality project. Software metrics discussed here are mathematically defined as a mapping from a code complexity to a numeric value. The implementation of the set of code metrics generation is called the code metrics calculation tool. [1] With the advantage of a metrics calculation tool, developers or researchers can assess calculate the metrics automatically on some extracted entities from the specific software projects. These metrics then can be combined to build a software quality model, which is greatly helpful for software quality analysis[5].

So far, there are many categories of code metrics calculation tools available online currently. It seems that developers and researchers can archive their desired code complexity metrics without modification by using these tools. However, after conducting a user study, we find most tools are of serious drawback on different aspects[1], especially for the calculation of Object-oriented language such as Java.

To make developers' life easy, enlightened by other researchers' product[6], we decide to build an efficient and accurate code metrics calculation tool that implements a variety of metrics calculation. The relatively difficult part of our task is considered to be the testing process of the tool and the evaluation of the tool. Testing process is the significant part that we need to validate the calculation correctness of JOnion. For this purpose we need the corresponding test suite and test cases. Since some of the metrics are complex to calculate, an amount of different test cases are required necessarily for testing the correctness of calculation on as many conditions as possible. Therefore, the number of test cases for a satisfiable coverage is huge. The last part is the evaluation of the metrics tool, which also can be challenging. The ideal way for JOnion evaluation is to compare with an "oracle" which has all the correct answers. However, there is no oracle available for code metrics calculation tool evaluation and we can just estimate the accuracy of our answers with comparison to other available metrics calculation tools. As to the comparison, the difference between different tools including JOnion will be analyzed and explained by choosing three canonical open source software projects.

II. Tool development

A. *Previous work*

The basic architecture of JOnion builds based on a previous preliminary program, namely "AssignmentParser", developed for the assignment of Mining Software Engineering Data course. This prototype was written in Java language using the API of an external Java library called JavaParser. JavaParser is a lightweight and easy to use Java parser with many desirable features including the abstract syntax tree

(AST) recording and tracing, visitor recursive support, and the capability to alter AST nodes or create new ones to modify the source code. This prototype already has been capable of some basic code metrics calculation correctly. However, it suffers many limitations on code structure, functionality and scalability, which cause serious problems on metrics extension. Therefore, our next work is to enhance the previous version of the "AssignmentParser" by implementing more practical metrics like object oriented metrics or other metrics. At the same time, we intend to re-organize the code structure of entities and components in the program and make it possible to generate metrics reports at different levels. (e.g. class, file, package, project)".

B. *Current work*

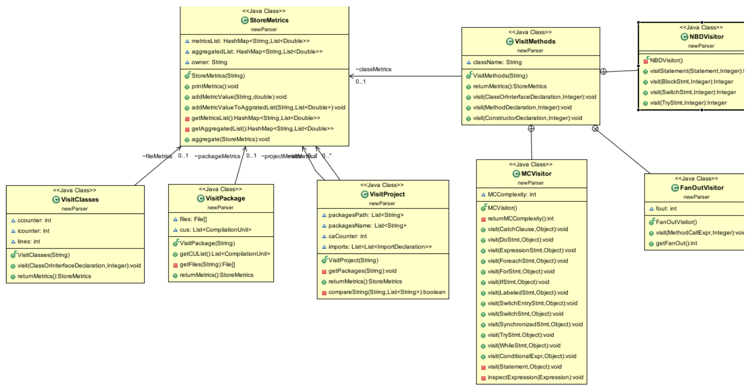
Because we adopt the approach mentioned in the bug mining paper, an general, elegant and robust aggregation method is the first task at the beginning of JOnion project. In the prototype "AssignmentParser", only unwrapped data structure exists for metrics data holding. And the worse is that there is no general aggregation method for all level. Aggregation methods have to be implemented in every level specifically. This handling is acceptable if only two levels involved in. But it makes a big barrier on level extension and contributes unpredictable errors during metrics extension. To deal with this situation, a new data structure called "StoreMetrics" is camp up with. "StoreMetrics" wraps up not only the data calculated in current level but also the aggregated data in the previous level and the "owner" of metrics that namely is the corresponding file, package or project name of current level.

With the above new data structure adopted, the aggregation method would be smoothly written down naturally. Technically speaking, in this method, taken the lower level code metrics lists extracted from the given "StoreMetrics" as inputs, calculate total value, average value and maximum value from the inputs and then store the results into the aggregated metrics list of "StoreMetrics" at the current level.

To organize the hierarchy clear and gain the high scalability of JOnion, we implement all metrics calculation at the corresponding level and store it in a data structure. At the same time, the general aggregate method implemented at the same level for the lower level metrics summary.

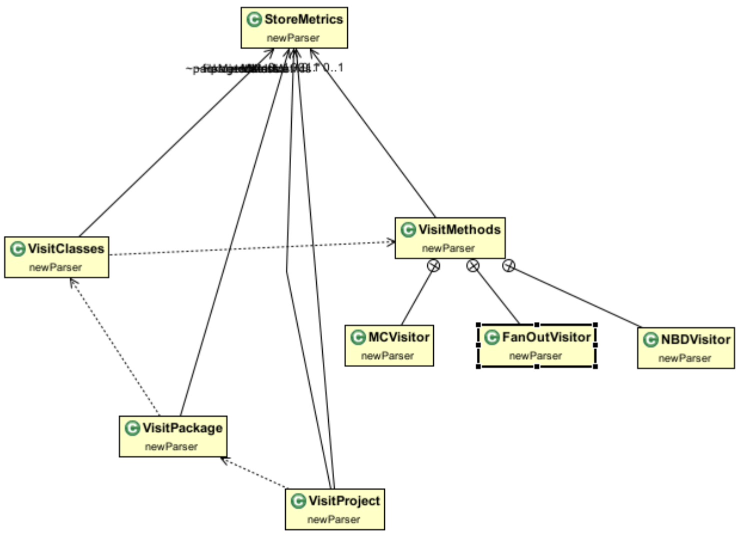
For example, all method level code complexity metrics such as "Number of method line" and "Number of nested block depth" are calculated at "VisitMethods" class and store them in the "MetricsList" of a created "StoreMetrics". No aggregation method implement at this level. Let's turn our eyes to the upper level, in the "VisitClasses" class, we calculate all class level code metrics and store them as well. In additional, the aggregation method for method level metrics summary is implement and the results store in "AggragatedList" of the given "StoreMetrics".

The below two diagrams indicate the overlook of JOnion code architecture. Graph 1 shows the entire system with methods detail. Alternatively, graph 2 shows the simplified system structure without involving in methods detail.



<<Java Class>>
StoreMetrics
newParser
metricsList: HashMap<String,List<Double>>
aggregatedList: HashMap<String,List<Double>>
owner: String
StoreMetrics(String)
printMetrics():void
addMetricValue(String,double):void
addMetricValueToAggratedList(String,List<Double>):void
getMetricsList():HashMap<String,List<Double>>
getAggregatedList():HashMap<String,List<Double>>
aggregate(StoreMetrics):void
~classMetrics
0..1
~fileMetrics
0..1
~packageMetrics
0..1
~projectMetrics
0..*
<<Java Class>>
VisitMethods
newParser
className: String
VisitMethods(String)
returnMetrics():StoreMetrics
visit(ClassOrInterfaceDeclaration,Integer):void
visit(MethodDeclaration,Integer):void
visit(ConstructorDeclaration,Integer):void
<<Java Class>>
NBDVisitor
newParser
NBDVisitor()
visitStatement(Statement,Integer):
visit(BlockStmt,Integer):Integer
visit(SwitchStmt,Integer):Integer
visit(TryStmt,Integer):Integer
<<Java Class>>
VisitClasses
newParser
ccounter: int
icounter: int
lines: int
VisitClasses(String)
visit(ClassOrInterfaceDeclaration,Integer):void
returnMetrics():StoreMetrics
<<Java Class>>
VisitPackage
newParser
files: File[]
cus: List<CompilationUnit>
VisitPackage(String)
getCUList():List<CompilationUnit>
getFiles(String):File[]
returnMetrics():StoreMetrics
<<Java Class>>
VisitProject
newParser
packagesPath: List<String>
packagesName: List<String>
caCounter: int
imports: List<List<ImportDeclaration>>
VisitProject(String)
getPackages(String):void
returnMetrics():StoreMetrics
compareString(String,List<String>):boolean
<<Java Class>>
MCVisitor
newParser
MCComplexity: int
MCVisitor()
returnMCComplexity():int
visit(CatchClause,Object):void
visit(DoStmt,Object):void
visit(ExpressionStmt,Object):void
visit(ForeachStmt,Object):void
visit(ForStmt,Object):void
visit(IfStmt,Object):void
visit(LabeledStmt,Object):void
visit(SwitchEntryStmt,Object):void
visit(SwitchStmt,Object):void
visit(SynchronizedStmt,Object):void
visit(TryStmt,Object):void
visit(WhileStmt,Object):void
visit(ConditionalExpr,Object):void
visit(Statement,Object):void
inspectExpression(Expression):void
<<Java Class>>
FanOutVisitor
newParser
fout: int
FanOutVisitor()
visit(MethodCallExpr,Integer):void
getFanOut():int

Graph 1. Entire UML diagram of JOnion



Graph 2. Simplified UML diagram of JOnion

However, not all of code metrics can be calculated following the above code structure. The big exception occurs during Afferent Coupling (CA), Efferent Coupling (CE) and Instability (I). All three code metrics calculations are implemented at project level because the information of adjacent packages within the same project is needed for the calculations but actually, these three metrics are belong to package level. So a routine to putting the calculation results back is built sspecially at "VisitProject" class, namely, class level implementation.

C. *Code complexity metrics*

The set of metrics that is primarily defined for our study originally comes from a study on Eclipse bug data. [2] In this paper they calculated the metrics at method level and used aggregation methods to build the reports at higher levels, up to file level. JOnion adapts this reasonable approach and extends this aggregation up to project level. Meanwhile, all of code metrics introduced in this paper have been included in JOnion. The code complexity metrics calculated in “AssignmentParser” listed in Table 1:

		Metric	File level	Package level
methods	FOUT	Number of method calls (fan out)	avg, max, total	avg, max, total
	MLOC	Method lines of code	avg, max, total	avg, max, total
	NBD	Nested block depth	avg, max, total	avg, max, total
	PAR	Number of parameters	avg, max, total	avg, max, total
	VG	McCabe cyclomatic complexity	avg, max, total	avg, max, total
classes	NOF	Number of fields	avg, max, total	avg, max, total
	NOM	Number of methods	avg, max, total	avg, max, total
	NSF	Number of static fields	avg, max, total	avg, max, total
	NSM	Number of static methods	avg, max, total	avg, max, total
files	ACD	Number of anonymous type declarations	value	avg, max, total
	NOI	Number of interfaces	value	avg, max, total
	NOT	Number of classes	value	avg, max, total
	TLOC	Total lines of code	value	avg, max, total
packages	NOCU	Number of files (compilation units)	N/A	value

Table 1. Code complexity metrics calculated in “AssignmentParser”

Except these code metrics mentioned above, moreover, JOnion provides user the access to other significant code complexity metrics such as CA, CE and I. The below table shows all metrics that JOnion is capable of computing:

abbrv. name	metric name	definition scope	calculation scope
FOUT	number of method calls	method	method
MLOC	method lines of code	method	method
NBD	nested block depth	method	method
PAR	number of parameters	method	method
VG	McCabe cyclomatic complxty	method	method
NOF	number of fields	class	class
NSF	number of static fields	class	class
NOM	number of methods	class	class
NSM	number of static methods	class	class
CLOC	class lines of code	class	class
ACD	number of anonymous type decl.	file	file
NOI	number of interfaces	file	file
NOT	number of classes	file	file
TLOC	total lines of code	file	file
NORM	number of overriden methods	class	class
WMC	weighted methods per class (sum of VG in class)	class	class
Ca	Afferent Coupling	package	project
Ce	Efferent Coupling	package	project
I	Instability (Ce / (Ca + Ce))	package	project

Table 2. Code complexity metrics calculated in JOnion

III. Software Testing

A. *Architecture of the testing system*

To test our tool JOnion efficiently and thoroughly, the famous test framework JUnit is adopted as the underlying test system.

JUnit is of several competitive advantages in testing area. First all, we are able to create a test suite to take fully control of all test cases running. It lets developers get rid of the tedious process that manually run each test case. Another advantage is JUnit allows developers to initialize the common settings shared among test cases at one setup method so that testers can focus more on the test cases themself. In terms of visualization, JUnit is also better than other test framework. After clicking “run” in Eclipse, JUnit would highlight the failed test cases by green bar and the passed test cases by red bar. At the same time, developers are able to see difference of the real results and the expected results if a failed status is received.

With taking the advantage of Junit, we created a test suite called “AllTest.java” which masters all of the developed test cases running in our test package and parallelly, developed all test case classes in the same package. All of above programs can be found in package “test”.

Except for the test suite and test cases, the corresponding sample Java programs used as inputs in each test case class are need as well. All of sample code is put under the “testcases” package.

As to testing certain special code metrics required multiple files located in multiple packages, such as CA, CE and I, we build another sample project called “TestedProject” aim to provide sample programs for the project level testing. A few of test cases in “test” will rely on this project as input.

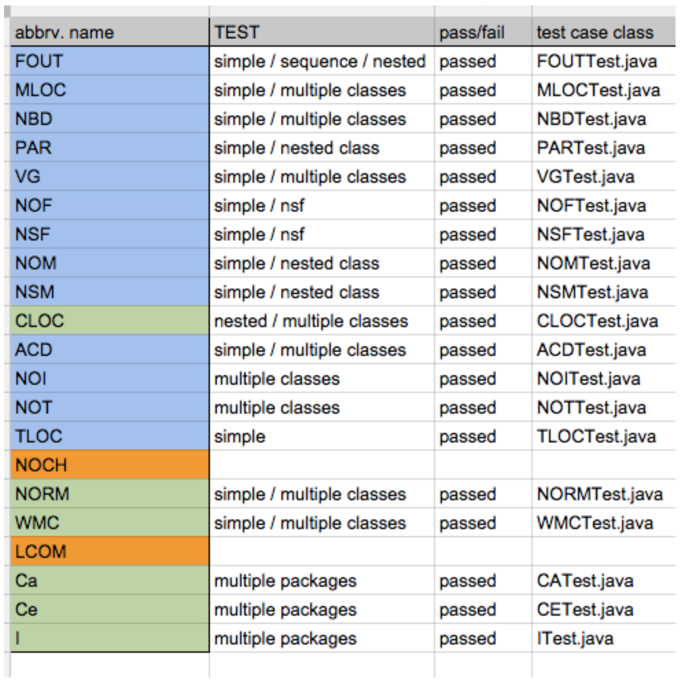
B. *Test cases and test coverage*

In each test case class, there are one or two test cases focused on the one particular code matrix testing only.

Same theory again, each sample test program cooperating with the corresponding test case is targeted for one code matrix testing only.

For example, in order to test nested block depth for a method, we design a test case class called “NBDTest” where two test cases sit and the corresponding test programs in “testcases” package.

One of test cases “nbdTest1_simpleTest” serves for testing the nested block depths of multiple methods in one single class. The input test program for this test case is “nbdTest1_simpleTest.java” in which there are various designed test methods or constructors that are of different nested block depth. We try our best to cover as many situations JOnion may deal with as possible. In “nbdTest1_simpleTest.java”, most of block statements such as “if” statement, “while” statement, “switch” clause and even “synchronized” block are involved in the methods or constructors multiple times. Table 3 lists all test cases, all relevant tested metrics and also all test case class.



abbrv. name	TEST	pass/fail	test case class
FOUT	simple / sequence / nested	passed	FOUTTest.java
MLOC	simple / multiple classes	passed	MLOCTest.java
NBD	simple / multiple classes	passed	NBDTest.java
PAR	simple / nested class	passed	PARTest.java
VG	simple / multiple classes	passed	VGTest.java
NOF	simple / nsf	passed	NOFTest.java
NSF	simple / nsf	passed	NSFTest.java
NOM	simple / nested class	passed	NOMTest.java
NSM	simple / nested class	passed	NSMTest.java
CLOC	nested / multiple classes	passed	CLOCTest.java
ACD	simple / multiple classes	passed	ACDTest.java
NOI	multiple classes	passed	NOITest.java
NOT	multiple classes	passed	NOTTest.java
TLOC	simple	passed	TLOCTest.java
NOCH			
NORM	simple / multiple classes	passed	NORMTest.java
WMC	simple / multiple classes	passed	WMCTest.java
LCOM			
Ca	multiple packages	passed	CATest.java
Ce	multiple packages	passed	CETest.java
I	multiple packages	passed	ITest.java

Table 3. Test plan and coverage table

IV. Other metrics tool

According to the importance of software metrics as we explained in previous sections, there exist lots of tools in both open source and commercial categories for calculating software metrics. All of them are of their own different advantage and disadvantage.

Consequently, big questions are naturally raised: how's the overall comparison of JOnion regarding to performance? What's the conceptual difference compared to other existing tools? Is JOnion tool competitive with them?

In order to compare the performance of JOnion with other existing tools of code metrics calculation, we should sample some of the other tools with similar functionality. For choosing the ones which are suitable for our purpose, there are some key points that we need to be careful about. First, the tool should be able to process java programs. Second, the set of metrics calculated by each of these tools may differ. So, we should select them with respect to the metrics we calculated in JOnion to ensure doing the meaningful comparison.

As a result, before evaluating the performance of JOnion, we decided to choose three representative open source code metrics calculation tools and investigate their characteristics. We also found a large amount of commercial calculation tools, but have no full access on all code metrics calculation. It is not convincing to just give partial metrics for comparison. So, in this paper, exclude them.

A. *Eclipse metrics plugin*

First calculation tool we investigated is called Eclipse metrics plugin 1.3.6 and 1.3.8, which are most popular online and are of the largest code metrics coverage among all available tools, whereas its limitation is obvious to see, that it can only worked in the eclipse environment and the worse is the code or project has to be completely built in the eclipse workspace without any error. If an error-existing or compile-impossible project requires to be analyzed, it would be an impossible task for Eclipse metrics plugin.

B. *Matrix++*

Second calculation tool we focused on is Matrix++, which is a lightweight static code analysis tool written in python language for C, C++ and Java. Comparing with Eclipse metric plugins, Matrix++ can analyze any code without compilation and no specific IDE environment required, just running in the command line. However, its coverage is relatively smaller than Eclipse metrics plugins'. Because of the generality on C, C++ and Java, no object-oriented metrics are calculated.

C. *CodePro AnalytiX*

Third calculation tool we next studied on is CodePro AnalytiX, which is a Java software testing plugin of Eclipse integrated with code metrics computation. His advantage is really impressive that a percentage chart will show up when clicking on a particular metrics after calculation done. Moreover, Its calculating speed is faster than Eclipse plugins'. However, it is Eclipse tool again and less object-oriented metrics generated.

V. Datasets

The approach used for the evaluation phase involves some critical analysis on the results of applying JOnion and three chosen tools on a set of practical software projects. So, the correctness and efficiency of the evaluation results largely replies on the reliability and representativeness of the chosen projects.

There are tons of software categories and also a wide variety of software applications in each of these categories. According to this fact, it is extremely important to choose carefully from all the possible options to have a neutral dataset. Therefore, the results of our study and our findings can be generalized to larger set of software systems. Our intention is to choose software systems from different categories in order to cover various characteristics of them and cover as many areas of software systems as we can. The sample software can be selected from the most popular repositories hosting services such as GitHub or SourceForge. These environments of hosting software deliver good categorization of their hosted softwares and all the source files of them are available to download. List the three typical sample projects chosen as dataset below:

Jaim implements the AOL IM TOC protocol as a Java library. The primary goal of JAIM is to simplify writing AOL bots in Java, but it could also be used to create Java based AOL clients. It consists of 46 source files, 1 package[1].
Jaim is a relatively small project, while it's a typical sample among the traditional standalone applications without GUI.

jTcGUI is a Linux tool for managing TrueCrypt volumes. It has 5 source files, 2 packages[1].
jTcGUI is a relatively simple complementary tool, obviously it's a typical sample among the frequently used utilities and complementary tools in a large commercial software.

ProGuard is a free Java class file shrinker, optimizer, and obfuscator. It removes unused classes, fields, methods, and attributes. It then optimizes the byte-code and renames the remaining classes, fields, and methods using short meaningless names. It consists of 465 source files, 35 packages[1].
ProGuard is a relatively large open source project with significant functionality. Therefore, it's a typical sample among the important milestone projects competing with commercial software.

VI. Evaluation

Besides design and implementation of the metrics tool, another important part of this project is doing an analytical comparison between our tool and other available software tools. The quantity and quality of metrics calculated by each of them should be evaluated and in some cases the metrics should be manually checked to assure about the correctness of calculations.

According to the other metrics section, we picked three other metrics tools, which are very popular and available for free. Then, the metrics calculated on the data set containing of three test software projects and the results analyzed in comparison to JOnion's for the same project. In each of the subsections, one of the alternative tools is discussed.

A. Eclipse Metrics Plugin

First calculation tool we investigated is called Eclipse metrics plugin 1.3.6 and 1.3.8, which are most popular online and are of the largest code metrics coverage among all available tools, whereas its limitation is obvious to see, that it can only worked in the eclipse environment and the worse is the code or project has to be completely built in the eclipse workspace without any error. If an error-existing or compile-impossible project requires to be analyzed, it would be an impossible task for Eclipse metrics plugin.

Lines of Code: In JOnion metrics tool, there are three metrics regarding lines of code, method lines of code (MLOC, method level), class lines of code (CLOC, class level), and total lines of code (TLOC, file level). The method we calculated lines of code is somehow different than lines of code are usually considered. Before counting LOC, the actual input code is preprocessed to create an intermediate code on which all the metrics are calculated. In this mid-level code, all the comments and black lines are removed from the code and also code commands that are broken into multiple lines to enhance readability would combine into a single-line command. For instance, in the Fig. 1 it is shown how this combining process works on an example. Therefore, the mid-level code used to calculate the metrics is more compact than the actual code and the metric, which reflects it, is the lines of code. Since most of the other tools calculate LOC based on the actual code of the program there is a noticeable difference between our value and theirs for this metric. Our method is based on a different approach to this metric, which provides information about real count of commands written in a piece of code.

There are some cases when the method lines of code calculated by both are equal like the class BuddyList.java in JAIM project. Also, in some other cases the value calculated by JOnion, is smaller than the Eclipse plugin like in LoginScreen.java in JAIM project because of the reason discussed. The same scenario is true for CLOC and TLOC.

```
try {
    UIManager.setLookAndFeel(
        UIManager.getSystemLookAndFeelClassName());
} catch (Exception ex) {
  System.out.println("Unable to load native look and feel");
}

try {
    UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
} catch (Exception ex) {
    System.out.println("Unable to load native look and feel");
}
```

Fig 1. Combining actual lines of code to one command per line version

Number of Fields and Methods in a Class: The number of fields and methods calculated by the Eclipse plugin are in most cases smaller than the value calculated by our tool. This difference is because some part of the code is not considered by the plugin. If there is an anonymous class declaration in the code, the fields and methods inside it are not considered by the plugin. While, JOnion also counts them in as the fields and methods of the class. Some examples for these metrics are shown in the Table 4.

	JOnion	Eclipse Metrics Plugin	More details
NORM	0	1	in JAIM project
NOF	9	7	in BuddyList.java in JAIM project
NOF	5	3	in RSA.java in project JAIM
MLOC	69	69	in BuddyList.java in JAIM project
MLOC	60	90	in LoginScreen.java in JAIM project
NOM	9	5	in BuddyList.java in JAIM project
TLOC	590	625	in JAIM project
PAR	58	not provided	in JAIM project
CE	0	7	in JAIM project
VG	13	8	in BuddyList.java in JAIM project
NORM	1	0	in Gui.java in JtcGUI project
NBD	3	4	in Password.java in JtcGUI project
CA	1	1	in JtcGUI project

Table 4. Analysis of Eclipse Plugin vs. JOnion for projects in dataset

Number of Overridden Methods in a Class (NORM): The value for this metric is checked for two of the projects in our data set manually and in both cases, it turned out to be calculated wrongly in by the plugin. On the other hand, The correct answer was provided by JOnion. In the JAIM project, the plugin's answer is one, while, the correct answer is zero. According to the plugin, the one overridden method exists in RSA.java file. In fact, there is no overridden method in this class and our tool's response is zero. In the second project, JtcGUI, the difference is more than the first case. The NORM values from the plugin and our tool are consecutively 14 and 29. All the difference caused by overridden methods inside anonymous class declarations that are not counted by the plugin. For example, in Gui.java class, there is one overridden method of the same kind but is not counted and value from plugin is zero for this class.

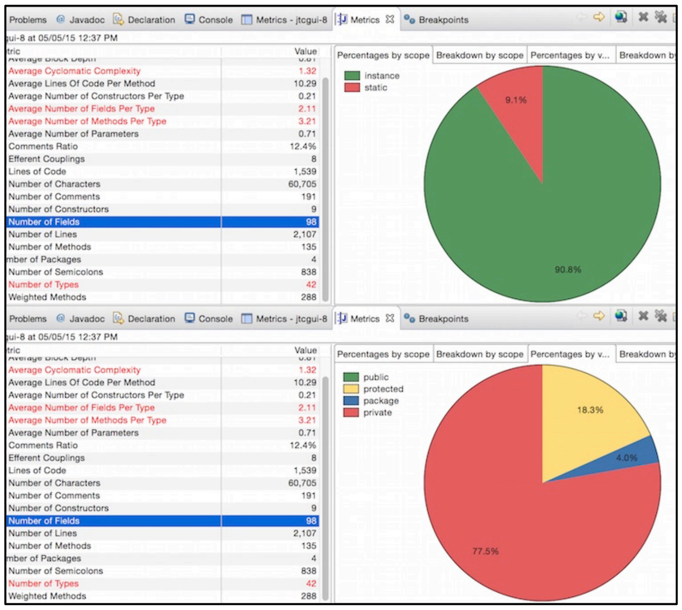
Efferent Coupling of a Package (CE)**:** Efferent Coupling is defined as number of classes inside a package, which have dependencies to the classes outside of that package. The CE metric which, is calculated by JOnion, is just focused on the relations and dependencies among the classes inside the target project. Hence, if there are dependencies to external classes or libraries, they are ignored in the calculation. Although, these dependencies are counted by the plugin and when the values for CE are compared, usually the value from plugin is greater than JOnion's. Our approach for calculation of CE also affects the value of Instability (I) since CE is used in the formula for calculating I.

Other Metrics**:** Some other metrics also have different values from the plugin output. We can mention McCabe Cyclomatic Complexity (VG) and Nested Block Depth (NBD) among them. The differences in calculated values are because of the anonymous class declarations, which are not considered in the calculation. But for NBD the cause is in the reverse direction since the anonymous class declarations in the code are not checked for calculation of NBD. Ignoring them for computation of NBD is the reason of difference in the value for this metric.

B. *CodePro AnalytiX*

The CodePro AnalytiX is another plugin for Eclipse to calculate software quality metrics. This tool provides smaller range of metrics than the Eclipse Metrics Plugin. The metrics can be calculated on any of the projects in Eclipse's workspace directory and the metrics are presented with either the average or sum value at the highest level. Also, for each of the metrics there is a plot like pie chart or bar chart to show the metrics values graphically.

Each of the metrics can be expanded to see their values for lower levels like package level, class level or method level. Moreover, there are more than one tabs for some of the metrics that shows min or max value for the metric or different plots to show different aspects of the metric. For example in the Fig 2 two different tabs for number of fields show pie charts when the metric is grouped based on being static/instance or public/protected/package/private.



gui-8 at 05/05/15 12:37 PM

tric	Value
Average Block Depth	0.81
Average Cyclomatic Complexity	1.32
Average Lines Of Code Per Method	10.29
Average Number of Constructors Per Type	0.21
Average Number of Fields Per Type	2.11
Average Number of Methods Per Type	3.21
Average Number of Parameters	0.71
Comments Ratio	12.4%
Efferent Couplings	8
Lines of Code	1,539
Number of Characters	60,705
Number of Comments	191
Number of Constructors	9
Number of Fields	98
Number of Lines	2,107
Number of Methods	135
mber of Packages	4
Number of Semicolons	838
Number of Types	42
Weighted Methods	288

gui-8 at 05/05/15 12:37 PM

tric	Value
Average Block Depth	0.81
Average Cyclomatic Complexity	1.32
Average Lines Of Code Per Method	10.29
Average Number of Constructors Per Type	0.21
Average Number of Fields Per Type	2.11
Average Number of Methods Per Type	3.21
Average Number of Parameters	0.71
Comments Ratio	12.4%
Efferent Couplings	8
Lines of Code	1,539
Number of Characters	60,705
Number of Comments	191
Number of Constructors	9
Number of Fields	98
Number of Lines	2,107
Number of Methods	135
mber of Packages	4
Number of Semicolons	838
Number of Types	42
Weighted Methods	288

Fig 2. Plot for metrics in the CodePro AnalytiX tool

Some of the metrics definitions are repeated in the metrics list but one of them is the average, min and max for the metric and the other one is the total number for the same definition. Therefore, the list of the metrics calculated by this tool is even shorter than it seems. Generally, when we check the values from this tool with the Eclipse plugin and JOnion, there are differences between all of them. It is not possible to say one of them is wrong because the differences in the values of the metrics are mostly originated from minor differences in the definitions and method of calculation. In the next parts, we explain more about each of the specific metrics and the differences between calculated values. For the specific comparison of metric values, the results for JtcGUI project are used.

Lines of Code**:** In CodePro Analytix, there are two metrics regarding lines of code, Average lines of code per method and Lines of code. The method lines of code values are similar to the Eclipse plugin but not exactly the same. Lines of code, which is similar to total lines of code in our definition, shows the total lines of code written for the project and it is the same as the plugin. All these numbers are different from the JOnion values because of the different approach to lines of code.

There are some more metrics regarding the lines of text in code files, one is Number of lines defined as total lines in the files of the project including blank lines. Some other metrics are defined to provide information about comments like number of comments and comments ration, which are unique to this tool and are not covered in other tools. In addition, there is another interesting metric called number of semicolons that has a more similar definition to lines of code in JOnion. Accordingly, the value for this metric, which is 838, is somehow near to 968 reported by JOnion for total methods line of code.

Number of Parameters**:** The max and average values for number of parameters given by all three tools are very similar. The only noticeable difference is among average values of this metric in the Eclipse plugin that is 0.64 and for the other two tools, it is 0.71 and 0.72.

Number of Classes: The number of classes for CodePro AnalytiX is defined as the total count of types in target elements in the project. This value is equal to sum of number of classes and number of anonymous type declarations in JOnion. For example, in the JtcGUI project it is 42 which is the sum of 8 (number of classes) and 34 (number of anonymous type declarations) in JOnion.

It also explains why the average value of class level metrics like number of methods and number of fields is dissimilar to JOnion and the Eclipse plugin.

Number of Fields and Methods in a Class: The total amounts for these metrics are exactly the same in JOnion and CodePro tool that is a reasonable outcome since anonymous type declarations are considered for the calculation of these metrics in both of the tools. In contrast, the maximum for number of methods are noticeably different between the mentioned tools. For JOnion it is 47 while the value of CodePro for this measure is 30. Interestingly, for the Eclipse plugin this measure is 31. Unfortunately, we could not explain how the calculation is done in the CodePro AnalytiX for the discussed measure. The only possible explanation is that there is a bug in the calculation for this measure.

McCabe Cyclomatic Complexity: The max value of this metric is 8 among all three metrics tools, which have been reviewed so far. However, when we look at the average value for this metric from the tools, they are slightly different from each other that is possible to be explained by the different number of methods

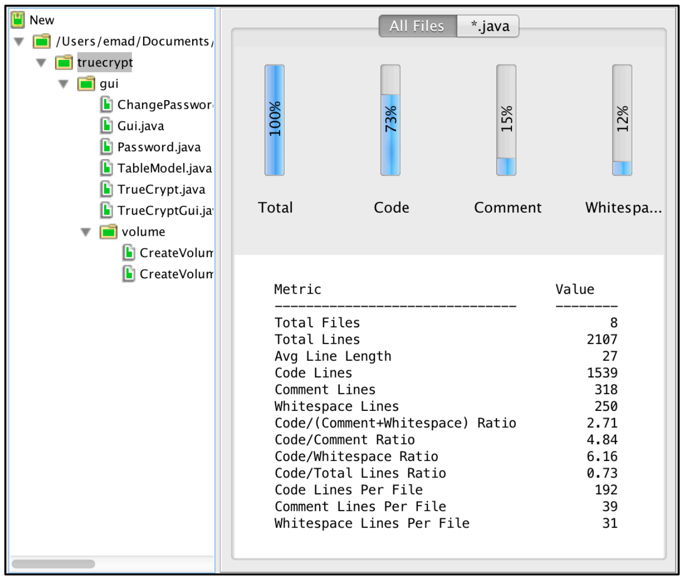
Other Metrics: Another metric that we can discuss here is the nested block depth. The max value from CodePro AnalytiX is the same as value calculated by JOnion. Although, Its average is neither similar to JOnion's nor the Eclipse plugin's. Once again, this can be explained with different total number of methods used by each of the tools.

All three tools also calculate weighted methods per class and efferent coupling. So, we can discuss about provided values by each of them in this part. First metric, that is sum of McCabe complexities in class, has the values 213, 177, and 288. These values are quite different, even though, it should be similar for JOnion and CodePro AnalytiX, since they are considering same group of classes, the difference is clearly significant. The same situation is true for CE measure and all the values are different from each other. For the class dependency in calculation of CE, external dependencies are considered as well as internal dependencies in both the Eclipse plugin and CodePro AnalytiX but still their results are different.

C. *Code Analyzer*

Code Analyzer is another alternative tool for code metrics calculation. Despite other two metrics tools, Code Analyzer is a standalone java application. This tool is highly focused on metrics related to lines of code, comments, and white spaces. Therefore, its metrics are limited to count of lines and ratios between different counts. An advantage of this tool, like JOnion, is that the target project should not necessarily be compiled for the metrics calculation and also this tool is capable to calculate these metrics for a variety of programming languages.

Code Analyzer presents the calculated metrics at the highest level which is the directory given to the application for calculation. It is possible to navigate through lower directories and files inside the input directory to see the values of metrics for them. In the Fig 3 a snapshot of the application running on JtcGUI project is shown.



```
Metric                                Value
--------------------------------      --------
Total Files                                  8
Total Lines                               2107
Avg Line Length                             27
Code Lines                                1539
Comment Lines                              318
Whitespace Lines                           250
Code/(Comment+Whitespace) Ratio           2.71
Code/Comment Ratio                        4.84
Code/Whitespace Ratio                     6.16
Code/Total Lines Ratio                    0.73
Code Lines Per File                        192
Comment Lines Per File                      39
Whitespace Lines Per File                   31
```

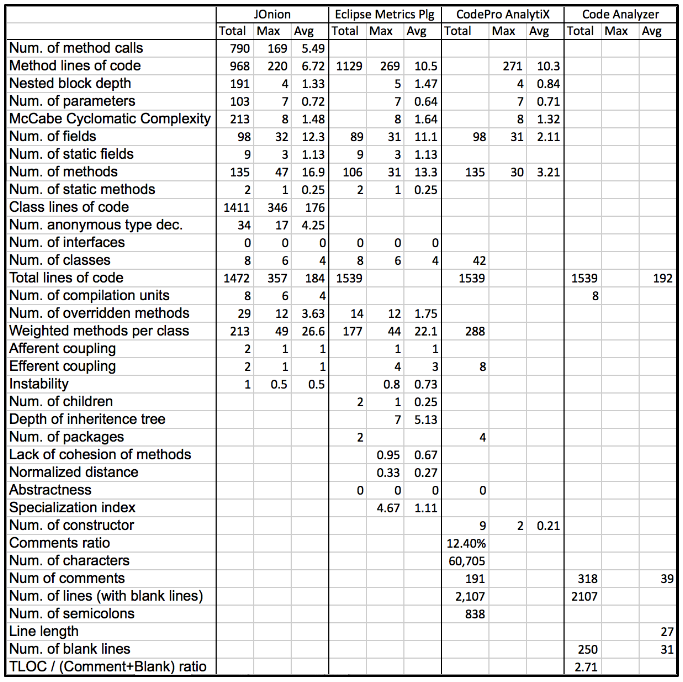
Fig 3. A snapshot of Code Analyzer metrics application

There are few metrics to discuss about in this tool because they are mostly about different line counts. First, number of files, or in another word compilation units, which is a very clearly defined metric. The number of files is 8 in the JtcGUI project, which is calculated correctly by both JOnion and Code Analyzer. Since in JOnion we have a layered perspective in the calculation of metrics, in addition to total number of files, we also provided max and average for number of files in the project level, because it is assumed a package level metric. Such approach to metrics is not observed in Code Analyzer tool.

The other metric, that we can discuss here, is total lines of code. The total value for this metric is 1539, which is equal to other two metrics tools tested. As it is explained before because of the different definition of line of code used in the JOnion, the values for this metric do not match with the result from other tools.

Finally, some other points about the Code Analyzer that worth mentioning: There was a unique metric among the set of metrics named average line length. This metric seems to be counted by number of characters but it is not clearly described. The number of comments is 318 which is totally different from 191 counted by CodePro AnalytiX. The reason is in the second one multiline comments are counted as one comment but the first one counts exact lines of comments.

Although, for reviewing last two metrics tools we focused on one of the projects in the dataset, JtcGUI, the results for other projects can also be similarly analyzed and discussed. The metric values for other projects in the dataset are provided in the Appendix of paper as well.



	JOnion			Eclipse Metrics Plg			CodePro AnalytiX			Code Analyzer		
	Total	Max	Avg	Total	Max	Avg	Total	Max	Avg	Total	Max	Avg
Num. of method calls	790	169	5.49									
Method lines of code	968	220	6.72	1129	269	10.5		271	10.3			
Nested block depth	191	4	1.33		5	1.47		4	0.84			
Num. of parameters	103	7	0.72		7	0.64		7	0.71			
McCabe Cyclomatic Complexity	213	8	1.48		8	1.64		8	1.32			
Num. of fields	98	32	12.3	89	31	11.1	98	31	2.11			
Num. of static fields	9	3	1.13	9	3	1.13						
Num. of methods	135	47	16.9	106	31	13.3	135	30	3.21			
Num. of static methods	2	1	0.25	2	1	0.25						
Class lines of code	1411	346	176									
Num. anonymous type dec.	34	17	4.25									
Num. of interfaces	0	0	0	0	0	0						
Num. of classes	8	6	4	8	6	4	42					
Total lines of code	1472	357	184	1539			1539			1539		192
Num. of compilation units	8	6	4							8		
Num. of overridden methods	29	12	3.63	14	12	1.75						
Weighted methods per class	213	49	26.6	177	44	22.1	288					
Afferent coupling	2	1	1		1	1						
Efferent coupling	2	1	1		4	3	8					
Instability	1	0.5	0.5		0.8	0.73						
Num. of children				2	1	0.25						
Depth of inheritence tree					7	5.13						
Num. of packages				2			4					
Lack of cohesion of methods					0.95	0.67						
Normalized distance					0.33	0.27						
Abstractness				0	0	0	0					
Specialization index					4.67	1.11						
Num. of constructor							9	2	0.21			
Comments ratio							12.40%					
Num. of characters							60,705					
Num of comments							191			318		39
Num. of lines (with blank lines)							2,107			2107		
Num. of semicolons							838					
Line length												27
Num. of blank lines										250		31
TLOC / (Comment+Blank) ratio										2.71		

Fig 4. Metrics results from all tested tools for JtcGUI

VII. LIMITATION

Basically, summarized from the above evaluation section, JOnion generally has competitive advantages compared with other existing calculation tools. However, the drawbacks of this version of JOnion exist evidently.

One drawback is that JOnion didn't provide the choice for these ambiguous metrics such that users can select their preferred matrix calculation definitions. The ideal scenario is that JOnion offers as many matrix calculation definitions as possible and then users can select free one definition for further calculation based on an excellent interpretation on definitions. For example, the matrix "number line of code" argued in the above section, it would be better to provide a neat choice instead of much argument.

In terms of metrics variety, although, the project level metrics calculation is much difficult than the lower level calculation, due to project level code metrics are extremely relevant with a project quality directly, more project level code metrics are needed to be introduced in JOnion.

Except for metrics calculation, the coverage of test cases and sample test code are not as large as we imagine. To ensure the quality of JOnion and widen the test coverage, more test cases and sample test code have to be created with different test view.

Turn focus to user experience of JOnion. As a calculation tools programming by Java language, a GUI interface based on Swing framework is required to serve more convenient user operation. At the meantime, once users receive the computation results, if a report export infrastructure exists, it would provide a better user experience. So, the report export system is what JOnion lacks currently.

VIII. CONCLUSION

Generally speaking, JOnion is a general efficient and accurate code complexity metrics calculation tools with a pretty wide coverage of metrics according to the summary of the evaluation section. Moreover, JOnion has high competitive strength in code complexity calculation market compared the open source calculation tools mentioned previously.

Therefore, for code complexity researchers, this version of JOnion has realistic meaning for code complexity investigation and would have a chance to acting a critical role in code metrics calculation tool area.

However, as discussed in above section, some drawbacks still exist in this version of JOnoion. On one aspect, in the future JOnion needs to enhance its weakness. On Another aspect, our future work will continue on efficient improvement as explored in the next section.

IX. FUTURE WORK

JOnion is still in the alpha stage with version number 0.25. A huge amount of work haven't done perfectly yet. Actually, At the beginning of this project or even in assignment period, the overall picture and prospect of JOnion has indeed been concerned by us.

As our tentative plan, at first, more meaningful code complexity metrics calculation especially project level metrics calculation would be implemented in the future. Due to time limitation, we did compute more complex metrics at project level than metrics in current version, but the complex calculation was such that we aren't able to done very well compared to other calculation tools.

Next improvement would be involved into the modification of the underlying framework JOnion relied on, Java Parser. More warning and error messages are urgently needed for a relatively large size development and also these messages must be clear and helpful for debugging.

If we can reach the JOnion 1.0 release stage, we hope that a user-friendlier version could be released in the future. A use-friendlier version could include a more easy-operated user interface and a report visualization system similar with Code ProX. In addition, an Eclipse plug-in version of JOnion also appears in our timetable.

ACKNOWLEDGMENT

We would like to thank greatly Silviu Musa, our partner during the assignment work. Without his positive participation and kind suggestions, we couldn't settle down our great decisions on JOnion and make the prototype "Assignment Parser" well in such short time.

At the meantime, we want to send our biggest thank to the course instructor, Professor Jack Jiang. We dropped by his office several times and ask him many questions during the office hour. Without his precise clarification on the project description and very professional suggestion on software engineering technique, JOnion was just staying at the prototype stage and the project might have chance to abort.

REFERENCES

1. Rüdiger Lincke, Jonas Lundberg and Welf Löwe. Comparing software metrics tools. In Proceedings of the 2008 international symposium on Software testing and analysis (pp. 131-142). ACM. 2008.

2. Thomas Zimmermann, Rahul Premraj and Andreas Zeller. Predicting Defects for Eclipse. In Proceedings of the Third International Workshop on Predictor Models in Software Engineering (PROMISE). 2007.

3. Kitchenham, Barbara A., Shari Lawrence Pfleeger, Lesley M. Pickard, Peter W. Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. Preliminary guidelines for empirical research in software engineering. Software Engineering, IEEE Transactions on 28(8). (pp. 721-734). 2002.

4. Javaparser. Retrieve from: https://code.google.com/p/javaparser/. 2015.

5. J. Muzamil. Software Metrics – Usability and Evaluation of Software Quality. Mater's thesis. University West. 2008.

6. Umamaheswari E., N. Bhalaji, D. K. Ghosh. Evaluating Metrics at Class and Method Level for Java Programs Using Knowledge Based Systems. ARPN Journal of Engineering and Applied Sciences (VOL 10, NO. 5, pp. 2047-2052)