2019-01-01

Code Smells Quantification: A Case Study On Large Open Source Research Codebase

Swapnil Singh Chauhan
University of Texas at El Paso, swapnil.utep@gmail.com

CODE SMELLS QUANTIFICATION: A CASE STUDY ON LARGE OPEN SOURCE RESEARCH CODEBASE

SWAPNIL SINGH CHAUHAN

Master's Program in Computer Science

APPROVED:

Omar Badreddin, Ph.D., Chair

Eric Smith, Ph.D.

Monika Akbar, Ph.D.

Yoonsik Cheon, Ph.D.

Charles H. Ambler, Ph.D.
Dean of the Graduate School

Dedicated to my inspiring parents and family

CODE SMELLS QUANTIFICATION: A CASE STUDY ON LARGE OPEN SOURCE RESEARCH CODEBASE

by

SWAPNIL SINGH CHAUHAN, B.E in Computer Science

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at El Paso

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE

Department of Computer Science

THE UNIVERSITY OF TEXAS AT EL PASO

May 2019

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Dr. Omar Badreddin, for the patient guidance, encouragement and advice he has provided throughout my time as his student. I have been extremely lucky to have a supervisor who cared so much about my work, and who responded to my questions and queries so promptly.

Besides my advisor, I would like to thank the rest of my thesis committee: Dr. Eric Smith, Dr. Monika Akbar, and Dr. Yoonsik Cheon, for their encouragement, and insightful comments. I would like to express my sincere gratitude to my advisor Dr. Hossain for his immense guidance and support.

I would also like to thank Dr. Jamie Acosta and Dr. Vladik Kreinovich for their help and encouragement to think more independently about our experiments, assignments and results. Apart from this, I will also like to give my word of thanks to my lab mate Rahad for his valuable suggestion which helped me during my thesis work. Last but not the least, I would like to thank my family for all their support.

ABSTRACT

Research software has opened up new pathways of discovery in many and diverse disciplines. The research software is developed under unique budgetary and schedule constraints. The developers are often untrained transient workforce of graduate students and postdocs. As a result, the software quality hinders its sustainability beyond the immediate research goals. More importantly, the prevalent reward structures favor contributions in terms of research articles and systematically undervalues research code contributions. As a result, researchers and funding agencies do not allocate appropriate efforts or resources to the development, sustenance, and dissemination of research codebases. At the same time, there are no uniform methodology to quantify codebase sustainability. Current methods adopt metrics with fixed thresholds that often do not appropriately characterize codebase quality and sustainability.

In this thesis, we conduct a case study to investigate this phenomenon. We analyze a large-scale research codebase over a five-year period. For reference, we compare the research codebase quality characteristics to a reference codebase developed by google engineers over the same period of time. The case study suggests that both research and professional codebases quality tends to degrade over time, but the decline is much more prominent in research codebases. Similarly, the study found that the number of code quality violations as a percentage of the codebase is significantly higher for the research codebase. The study also reveals that there are quality characteristics that are unique to professional codebases. For example, professionals tend to design software with smaller code units possibly in anticipation that such units will grow in size overtime. On the other hand, research codebases' units are significantly larger and become increasingly difficult to maintain over time. The results of this thesis are published in **2019** IEEE/ACM 14th International Workshop on Software Engineering for Science (SE4Science).

This thesis is organized as follows. Chapter 1 provides a background on code quality and presents a number of code quality metrics. This chapter also presents the two codebases used in the study. Chapter 2 presents related works pertaining to software engineers' perceptions of software quality, the evolution and impact of code quality metrics, methods to quantify code quality, and a review of key methodologies and tools to identify and quantify code quality. Chapter 3 presents the case study design and introduces the quantification metrics developed for the study. Chapter 4 presents the results, analysis, and key findings. We conclude the thesis and outline future work in Chapter 5.

TABLE OF CONTENTS

LIST OF TABLES

Page

LIST OF FIGURES

Page

CHAPTER 1: BACKGROUND

Code smell is first officially introduced in book Refactoring: “Improving the Design of Existing Code” by the Martin Fowler in a year (1999). Code smells are not the same as language syntax error or other compilation warnings - compilers can discover those for us - but instead signs of terrible program plan or awful programming practices that could represent an issue to you (or another software engineer dealing with the code sooner or later) when the program should be changed, for an example suppose you have added a new functionality to existing module and you found troublesome code then these codes could be changed by applying a refactoring. We say 'could', claiming few out of every odd code smell implies there is an issue.

1.1 Introduction to Code Smell

In Software development, Code Smell is often considered as a measurement of the software quality. It is an indication that few sections of the code need to look again either for refactoring, optimization or improving performance. There is a variety of code smell that can be seen in code that can be referred to the symptoms indicating problems with the code. We should never forget that the context of writing the specific code is important, sometimes detection of code smell may refer that code need to analyze again for optimization, but it can be acceptable in its present form if it performs designated task efficiently.

As we all know that Software maintenance is expensive, sometimes to make a small change in the module functionality the developer ends up changing the lots of code which in then increase the overall cost of the project. By identifying the code smell, early in the software life cycle could reduce excessive cost and delay of fixing them later.

Code smell is easy to Sense, and it is being observed most of the performance issue are caused by code smell. For an example Writing a huge program in terms of length of code can cause performance issue because for using the one functionality, we need to traverse whole and that would end up consuming memory, elapse (total running) time and purposed a code refactoring. A badly written code can cost huge in term of maintaining it in the long run. In large code base, there are many developers who made the changes on the same program, so it is important to follow the coding standards in order to improve the code readability and efficiency.

Besides that, refactoring technique can utilize to influence source code simpler to peruse and change. Before any refactoring tool, the software developer would decide the area that is needed to be refactored.

1.2 Code Smell Category

There is a variety of code smell and each one of them has their specific definition. There are few smells which we have considered during our experimentation like Large/Small Class, Large/Small Method, Large/Small Parameter List, Code to comment ratio, Extensive use of Switch Statement, Feature Envy, Poor Naming convention, and duplicate codes. Short description of various code smell is listed below:

Table 1.1: Code Smell Category

Code-Smell Reference	Description
Large Class [17]	The Large class is a class which is trying to do too much, contains many variables, function or in terms of line of code. It makes class more difficult to read, understand or trouble shoot in case of

	any error. There might be the possibility of refactoring the large Class into smaller one.
Large Method [18]	The Large function is considered to be long when it possesses many Line of code, and that is why it is termed as complex and difficult to understand the exact role of a function. There might be the possibility of refactoring the large method into smaller one.
Large ParameterList [18]	The number of the parameters passed to the function/methods. The large number of parameters worsen the code quality and readability of function. It is an indication that a method possesses enough functionality and need to be refactor.
Feature Envy[18]	The method of a Class that uses feature of other class more than its own. It is better moves the method to other class might help.
Code to Comment ratio	Comments plays a critical role in the programs. It provides the hint on the intent of the logic applied in code. But there is difference between comments that are useful and comments that are not. Having too much in the program may fail its purpose.
Small Method	The method literally does nothing. It should be merge with the other suitable method.
Duplicate codes [18]	Duplicate code is generally considered undesirable and Sequences of duplicate code are known as code clones.

Cyclic Hierarchy [21]	This Smell arises when superclass depends upon an of its subtype class.
Small Class [18]	A Class literally does nothing. It should be merged with other suitable class.
Standards like naming conventions (Percentage follow)	It determines whether a project follows the basic coding standards like proper variable, function and class naming convention. following the coding practice improve the code readability.
Middle Man [18]	A Class or method with no logic but act as an interface between two modules
Dead code [19]	A code that is no longer used in the program
Refused Bequest [18]	A class doesn't use things it inherits from its superclass
Inappropriate-Intimacy [18]	The smell occurs when a class often access the other class method which in turn represent highly coupled relation between two. It could be a chance that the both methods are serving same functionality.
Swiss Army Knife [19]	This type of a smell occurs when a developer tries to build too much of functionality in class make it more complex to understand.
Message Chains [18]	This smell occurs when function acquires the series of the object sending a message to each other.

Divergent Change [18]	This smell occurs when changes made by many developers to single class for different reason.
Shotgun Surgery [18]	Making one change requires changing code in multiple places
Obsolete Class [22]	The Smell occurs when a class is no longer used in system.
Assertion Roulette [20]	Multiple assertions in a test method with no explanations
Sensitive Equality [23]	Equality checks that are affected by nonessential things
Eager Test [20]	A test checks several methods of the object it's testing
Incomplete Library Class [18]	Libraries that don't provide all needed methods
Indirect Testing [20]	A test class testing objects other than the main one it is testing
Mystery Guest [20]	Tests using external resources
Speculative-Generality [18]	Making code more general in case it's needed later; Unused hooks and special cases make code more difficult to understand
Temporary Field [18]	Instance variables set only in certain circumstances or fields used to hold intermediate results
Test Code Duplication [20]	Code repeated in multiple tests
Primitive Obsession [18]	Using primitive data types where classes or record types would work better
Parallel-Inheritance Hierarchies [22]	Every time you make a subclass of one class, you have to make a subclass of another
Dependent test methods [23]	One test method depends on another to be able to run

Switch Statements [18]	Using a switch statement where polymorphism would work better
Lazy Class [18]	A class that isn't doing enough work to justify its existence
Data Clumps [18]	Sets of variables usually passed together in multiple places
Alternative Classes with Different Interfaces [18]	Two Classes performing the same task but it possesses different name

1.3 About QGIS and Tensor-Flow

The QGIS [33] project is an open source software and available over the Git-hub. Quantum Geographical Information System is a cross-platform free and desktop geographic information system (GIS) application that supports viewing, editing, and analysis of geospatial data.

It functions as a geographic information system (GIS) software, allowing users to analyze and edit spatial information, in addition to composing and exporting graphical maps. The project comprises of 1216368 lines of code, 6 primary modules/Package, 9307 functions.

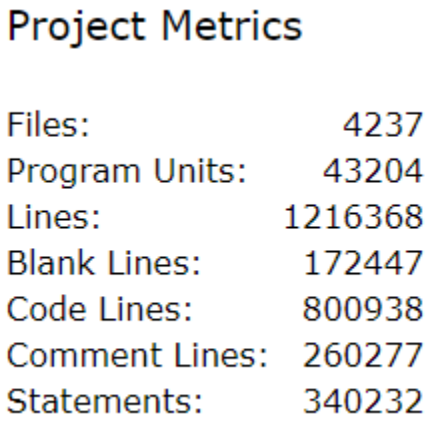


Fig 1.1: QGIS Code breakdown and Project Metrics from Tool Understand

There is a total of 1.34 million lines of code Present in QGIS [33]. The line of code is one of the widely used size metrics. We have calculated the total number of lines in each function, classes, modules. We have then categorized the code length for each language and also listed the blank lines and lines used for comments in the QGIS. Below is a table which list all the language used in the QGIS along with total LOC, Comment lines and Comment Ratio. Below is the detailed code metrics of QGIS extracted from the Tool Understand [31].

Table 1.2: QGIS Code Metrices – Tool UNDERSTAND

Language Used	Lines of code	Comments line	Comment ratio (%)	Blank lines	Total lines
C++	664808	167144	20.1	143290	975242
Xml	441041	45	0	1200	442286
Python	109622	25197	18.7	28.77	162896
C	64342	1160	1.8	590	66092
Qml	20984	0	0	33	21017
Cmake	12900	2010	13.5	2273	17183
Ampl	7589	3	0	0	7598
HTML	6071	6	0.1	568	6645
CSS	4388	27	0.6	56	4471
Shell Script	2581	1043	28.8	580	4204
Perl	2374	397	14.3	484	3255
XML Schema	1660	22	1.3	0	1682
SQL	920	74	7.4	181	1175

NSIS	371	131	26.1	145	647
OpenGLShading	193	38	16.5	46	277
DOS Batch Script	155	2	1.3	26	183
XSL Transformation	113	0	0	6	119
JavaScript	37	1	2.6	7	45
Objective C	33	48	59.3	16	97

Tensor-Flow [34] is an opensource library which is built by the Google brain team mainly written in the C++ and python. It is widely used in the machine learning application. It has more than 2 million lines of code. It is majorly written in C++ and python which are 48% and 38% respectively. It is an active and highly rated project with 117565-star rating on the GitHub. At present, there are 46379 commits and 79 releases.

Table 1.3: Tensor-Flow Code Metrices – Tool UNDERSTAND

Blank Lines	669077
Commented Lines	1036836
Total Lines	5709781
Lines of Code	3653717

1.4 Tool Selected for Input extraction

In order to extract the data for our experimentation, we have used the Tool – Understand [31] to generate the report on QGIS [33] and Tensor-Flow [34] to get us the length of Class, method, lines of comment which are further used in code smell measurement. Similarly, we have used PMD-CPD [32] to detect the code clones. Below is the information on the tools which we have used during our analysis.

1.4.1 Background on Static Analysis Tool - Understand

Understand [32] is a static code analysis tool mainly used for generating code metrics and could be used for performing the code review and standardized testing. It can be used to understand the large code bases with multiple languages. Below is the list which provides a brief overview of languages supported by the tool understand.

C, C++, C#, Objective C/Objective C++, Ada, Assembly, Visual Basic, COBOL, Fortran, Java, JOVIAL, Pascal/Delphi, PL/M, Python, VHDL, and Web (PHP, HTML, CSS, JavaScript, and XML).

Fig 1.2: Snapshot from understand – Supported Language

It has syntax coloring editor and offers code navigation by cross-referencing. It is available in Windows, Linux, MacOS and Unix. Understand gives you appropriate data in regard to your code. Rapidly observe all data on methods, classes, variables, and so on., how they are utilized, called, modified, and associated. Effortlessly observe call trees, measurements, references and some other data you would need to think about your code. Understand is exceptionally effective at gathering metrics about the code and giving distinctive approaches to you to see it. There is a considerable accumulation of standard measurements rapidly accessible and also choices for composing your own particular custom metrics when we don't cover precisely what you require.

It also offers charts and graphs that enable you to perceive how your code associates (conditional dependency), how it flows (control flow graphs), what functions call other functions (call graphs).

As part of reporting is concerned, it can generate various reports like the structural, cross reference, quality, metrics, and many custom based reports. It allows the user to visualize the data in form of the tree map.

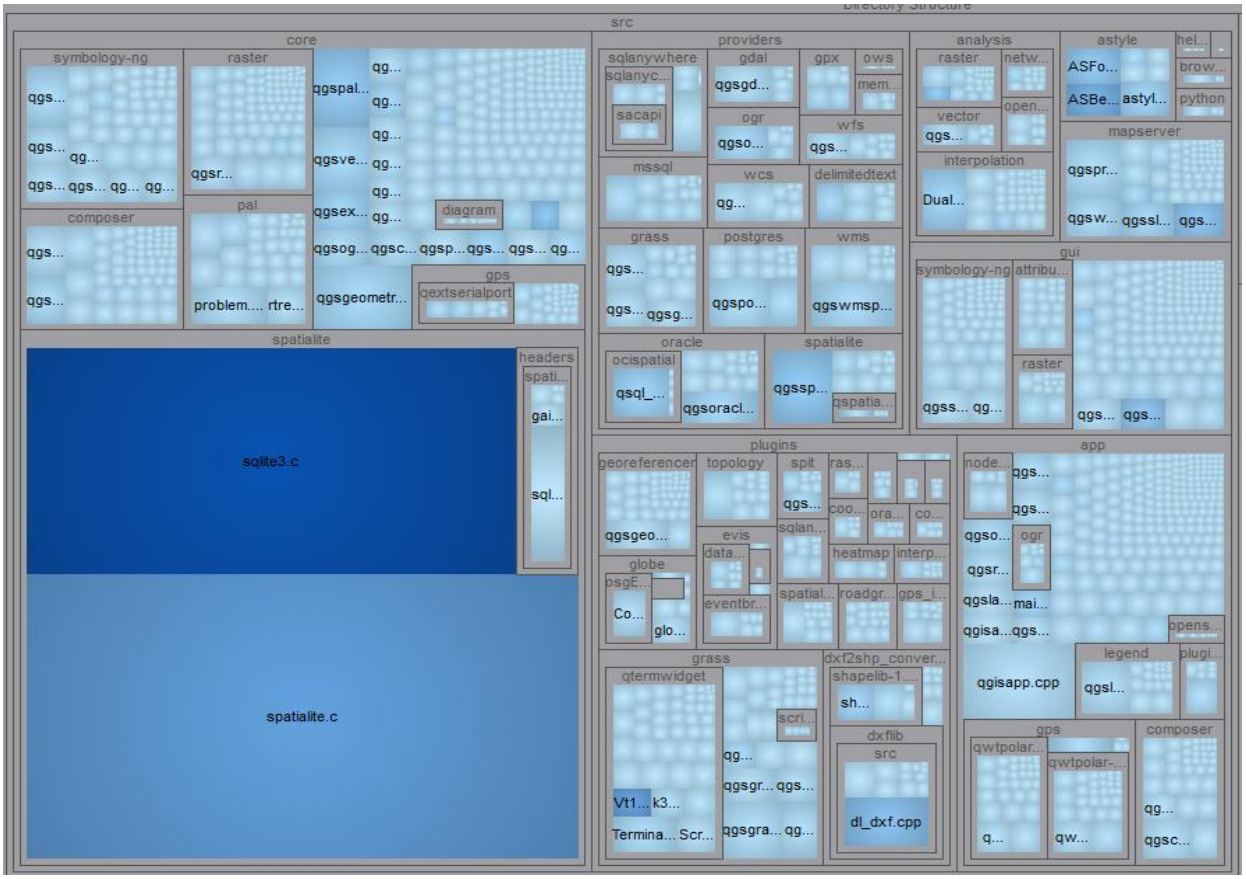


Fig 1.3: Snapshot from Tool understand – Heat map

It breaks down the code and makes a database of connections and relations. The data assembled amid that examination can be immediately found in the Information Browse.

1.4.2 Background on Clone Detection Tool - PMD-CPD

PMD-CPD [32] is a static source code analyzer. It could detect the duplicate code clones in Large software system and works with Java, C++, C, C# and many languages. It is available in GUI platform where it allows the user to manually select duplicate chunk size, language, include/exclude literals, identifiers, and annotation. It supports multiple report format like CSV, text, XML and also displays the result on the UI where code editor lists out duplicate clone location along with the matched files name and lines.

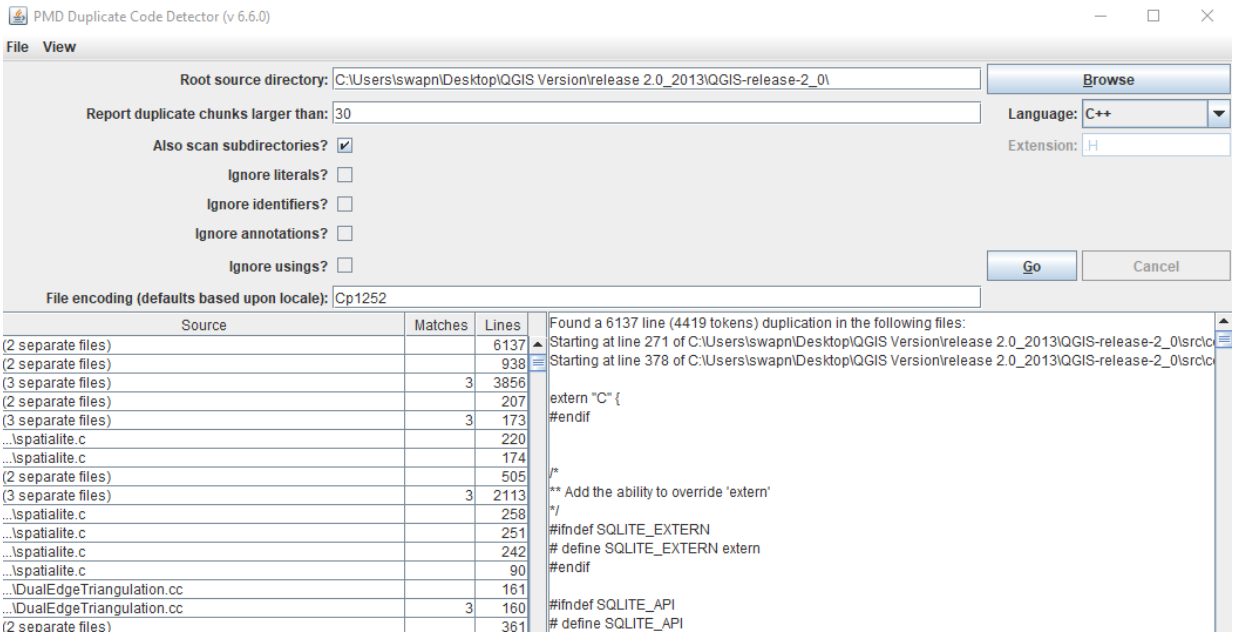


Fig 1.4: Snapshot from Tool PMD-CPD – Code Clones

On the off chance that you wish CPD [32] to parse an unsupported language, you can without much of a stretch build up another parser for CPD. Apart from this PMD also performs the static code analysis which discovers basic coding flaws like unused factors, void catch squares, unnecessary object creation etc. It basically works with Java and Apex, however, underpins six

different programming languages. It can be used for quality assurance of software in post development and testing phase for setting the standard for the codebase. It also allows the user to write the customary rules and build their own metrics.

CHAPTER 2: RELATED WORK

This Chapter talks about the work done in this area. We have listed the literature work indicating the developer perception on code smell.

2.1 Perception of Developer on Code Smell

Several survey studies have been done recently to understand the nature, impact and developer perception on the code smell. The survey performed by the M. V. Mantyla [6] studied the relationship between evaluating the code quality and other measures. This paper put up three research question to direct their research are as follow 1. Do, and if this is true, how, the experience and ability of designers influence the Code smell assessments? 2. Do programming designers have a uniform feeling on the foulness of the code? 3. Do the developers' assessments on code smells connect with related source code measurements? This paper has selected the Beach Park Finnish software product development company as Use-case. The beach park has developed 2 software in Delphi which is an extension of Pascal language. The Questionnaire was prepared in two parts. In the first part, the respondent has provided their background information such as age, role, education and work experience in the software company. In the second part of the questionnaire, they were asked to access 23 code smell on the module which they have primarily worked upon. 12 out of 18 developers participated in providing the response. As part of the result, they have observed that the utilization of smell for code assessment designs is hard because of the clashing perception of various evaluators. Secondly, In the survey, all code smell was assessed against every module the developer had worked with. This could cause predisposition on singular assessments in view of the nature of the other modules the developer

had worked with. There is no uniformity among developer on the code smell. For the final question, overall it creates the impression that developer assessments of the smells don't connect with the utilized source code measurements.

There is another paper written by A. Yamashita [7] which talks about the importance of code smell for software developers. If the developers don't really care about the Smell then, is this due to the absence of pertinence of the concepts, or because of the lack of awareness about code smells on the developer side, or due to an absence of code smell detection tools. The marketplace which is selected for the survey is "Free-lancer.com", 85 professional developers were selected to complete the survey, but 73 completed it completely. A survey Questionnaire is prepared which asked the respondent about roles, age, language expertise, working experience, popular code smells, code smell analysis tools, their awareness on the code smell and the perception of the usefulness in a 5-point ordered response scale. The data has been collected from various countries like India, USA, China, and 26 other countries. As the result from data gathered, Duplicate code, long method and accidental complexity are the high-ranking code smell. Secondly, on the perception of criticality of code smell, it varies as per the roles of the individuals like project managers are less concerned while the self-employed professional is much more concerned about the criticality. On the usage of code smell analysis tools, 70% professional said that they don't use any tool while 30% professional accepted that they have used one or more tools, but they majorly help in detecting the duplicate code. On the usefulness of code smell as per the data received, 29 developers say that it is moderately or extremely used for code inspection, 33 says that it is useful for error prediction and 17 answered for refactoring guidance.

Another paper written by F. Palomba [8] provide an empirical study to answer two research question: (1) To what extent developer see bad code smell as the design problem, and (2) what are those harmful code smell that developer thinks is most harmful? The author selected three open source software ArgoUML, Eclipse, and Jedit. In order to provider answer to the research question, they have invited the 15 Master students, 28 Industrial Developers and 45 developers working on above listed open source project. They have prepared the questionnaire available that all participants need to answer over a web application. To answer the first question, for each type of the code smell, identify the percentage of the participants considered it to be bad smell and percentage of time that specific smell is identified in the open source software. As a result, they have come to the conclusion that Smell generally not Perceived as Design or Implementation Issues and there is smell whose perception may vary.

Katzmarski [13] investigate whether metrics agree with complexity as perceived by programmers. Their findings point programmers' opinions are quite similar and only a few metrics and in only a few cases reproduce complexity rankings similar to human raters. their work focusses on the influencing factors instead of analyzing the metrics associated with code smells.

Another paper written by T. Vale and I. S. Souza [12] talks about the developer's perception on code smells, factors that influence code smell and how to manage these factors. For this purpose, they have performed three case studies by interviewing software developers from three different real-world systems. After the data collection, the author has performed cross-case verification. As a result, it is been established that each developer has a different level of the understanding of the code smell, there are several influencing factors like short deadlines, bad architecture design decision, lack of business knowledge, lack of experienced developers and

on how to manage these factors many developers have suggested the following strategies: additional time for refactoring in the development schedules, a higher priority for software quality, better understanding of the adopted technologies and training for inexperienced developers.

2.2 Evolution and Impact of the Code Smell

There are few studies which talks about the evolution and impact of the code smell. Paper written by S. Olbrich [1] is one of them which investigate the two-code smell [God Class and shotgun surgery] by analyzing the historic version data of two large open source system software. The information provided in this paper is useful for identifying the critical areas for refactoring in order to secure positive software evolution. The research question addresses the number of code smell changes over a long period of time in the software if no measures were taken to improve the software quality. In order to answer a question, they have selected Apache Lucene and Apache Xerces 2 Java (both the opensource software developed by Apache foundation). As per data collected it is been observed that the number of god class has increased when they compared the latest version with the oldest ones. The results show that we can identify different phases in the evolution of code smells during the system development and that code smell infected components exhibit a different change behavior.

Another paper written by A. Chatzigeorgiou [9] which uses tools to identify various code smell in the code to explores the presence and evolution of such problems by analyzing past versions of code. They have selected the following question in their research paper. (1) Does the number of design problems increase over time? (2) Will the evolution of a software system remove some of its bad smells or are the problems solved only after targeted maintenance

activities? (3) Do bad smells exist in a software module right from its initial construction or do they appear during its evolution? (4) How frequent are refactoring activities that target bad smells? (4) How urgent is it to remove the identified code smells?

The code smell detection tool Jdeodorant is used to analyze package of JFlex (which is lexical analyzer generator of Java) comprise of 30 classes and package of JFreeChart consisting of 110 classes. The bad smell is being identified in 10 version of JFlex and 14 version of JFreeChart out of which long method code smell is one of largest identified smell. It is being observed that the number of smells increased as a new version has come up. To further answer the question, the author has created 5 categories namely, A (Smell existed in all the version), B (Smell identified in one version and resided till latest version), C (Smell identified in first version but now it disappeared), D (Smell identified in previous version but now it disappeared). As per the observation, in JFlex there are 90% of cases are identified under A and B Category and 10 % of cases identified under (C+D) category while in JFreeChart, 78.8% of all cases (A+B) and rest of percentage. The author has built average time persistence metrices which shows that Long method and feature envy code smells persist for much longer time. Based on data they have built a grey block to represent the percentage of the active smell residing in software. The results indicate that in most cases, the design problems persist up to the latest examined version accumulating as the project matures. Moreover, a significant percentage of the problems were introduced at the time when the method in which they reside to added system.

Another similar paper written by F. Khomh [10] which the author investigates if classes with code smells are more change-prone than classes without smells. In order to test this hypothesis, they had 29 code smells in 9 releases of Azureus and in 13 releases of Eclipse and study the relation between classes with these code smells and class change-proneness. As a

result, all releases of Azureus and Eclipse classes with code smells are more change-prone than others.

2.3 Quantification and Assessment of the Code Smell

The first technique to quantify the code smell is given by S. Olbrich [1], this paper mainly focused on detecting the two-code smell which is god class and shotgun surgery. The paper quantifies the god class based on the (1) weighted method count which is nothing but the static complexity of all method presents in the class is greater than or equal to 47, (2) tight class cohesion > 1/3 and (3) total counts of foreign attribute which are accessing the class in some way is greater than 5. The set of metrics used for measuring the shotgun surgery is based on changing method (number of methods that called other methods in class > 10) or changing class (count of class in which method calls other methods> 5). The threshold is based on the statistical measurement which is often used in the metrics related to size as mentioned the book written by Marinescu, Radu “Object-Oriented Metrics in Practice” where space is portioned into meaningful threshold value.

The Second approach in this area to measure the feature Envy code smell is given by Lanza and Marinescu is based on the following metrics: (1) methods which are using the function of some other class, or (2) Count of the methods which are using other class functionality more than their own class, or (3) Foreign attribute measurement by the FDP metrics. Similarly, Mäntylä has suggested using polynomial metrics which means to combine NLOC (Number of lines of code, cyclomatic complexity, and Hallstead metrics) to quantify the Long method code smell.

Another paper “DETECTION AND REFACTORING OF BAD SMELL CAUSED BY LARGE SCALE” written by Jiang Dexun. In this paper, he identifies the bad class location in the file based on the inner cohesion of that particular class. In order to quantify the large class code smell, author introduce a cohesion metric which is the rate of the average of the distance of entities out of the class and those in the class.

$$Conesion_C = \frac{\frac{\sum_{e_i \notin C} Distance(e_i, C)}{|e_i \notin C|}}{\frac{\sum_{e_i \in C} Distance(e_i, C)}{|e_i \in C|}}$$

If the cohesion value is small that means it has low value of the cohesion. So, the classes with a low cohesion metric value could be identified as large classes. The low cohesion value indicates the large class code smell.

Finally, there is a paper authored by the S. M. Olbrich [1] where he used an approach to detect the brain method code smell. Below is the formula used for brain method code smell detection. The formula uses the length of the code, cyclomatic complexity, nested level in the method and number of the variable into consideration along with their threshold limit.

$$BM(M) = \begin{cases} 1, & ((LOC(M) > 65) \wedge \\ & (CYCLO(M)/LOC(M) \geq 0.24) \wedge \\ & (MAXNESTING(M) \geq 5) \wedge (NOAV(M) > 8)) \\ 0, & else \end{cases}$$

Code smells detection tools like PMD uses to a threshold value for measuring the code smell. These tools merely provide the list of classes and method which lies within or outside the specific set value, but they have no clue on the severity of the detected code smell. It is also

important to measure how bad is the code smell. I believe, we need to penalize the class or method which are away from the set threshold value.

2.4 Tools Techniques and Approaches to Identify the Code Smell

Several studies are performed on the code smell detection tools and their approaches to detect the specific type of the smell, below are few listed papers which talks about them.

There is a conference Paper written by F. A. Fontana [14]. The aim of this paper is to describe the author's experience in using different tools for code smell detection. The paper briefly introduced the principal code smell detection tools available in the market and describe the experiments they did with these tools on several systems. The Author has used the Gantt Project for the case study and provided validation and a comparison between various tools. In order to detect the code smell, they have used DÉCOR benchmark (Large class – LOC > 415 & Large method – LOC > 48 & Large Parameter list – 4 or above). Various Tools (PMD, Jdeodrant, iPlasma) have been listed under this paper along with their ability to detect the code smell.

Another Paper written by the Liu, Xinghua & Zhang, Cheng [15] talks about the detection metrics which is integrated with the code smell detection tools. researchers often put these metrics for measuring the smell. The tools like Checkstyle and PMD adopt these kinds of metric.

Both the tools have used a different kind of threshold values, Checkstyle use the threshold value of 2000 LOC and PMD uses 1000 LOC for large class code Smell. For detecting Large method, Checkstyle use threshold 150 LOC while PMD use threshold 100 LOC. The author also talks about the ways to measure the long parameter list smell, as per Danphitsanuphan et al. utilize NPM thought, they give a limit 5. A number of parameters in method (NPM). In the interim, these researchers identify long parameter list through distinguishing through the idea of Abstract Syntax Tree (AST) by tallying the quantities of the Single Variable Declaration (LPL) which is one point

of Abstract Language structure Tree (AST). Tools like PMD [32] and Checkstyle uses these detection metrics but have different threshold limit.

CHAPTER 3: CASE STUDY DESIGN AND CODE SMELL QUANTIFICATION

This Chapter describe our case study design and the approach to quantify the code smell in QGIS and Tensor-Flow (open source software). For our analysis, we have selected the few codes smells like Large class, Large Method, Small Class, Small Method, Small and Large Comment Ratio, Large parameter list and duplicate code. Code smells can be detected in the source code through various aspects. We have studied the evolution of code smell by applying the formula stated on different versions of QGIS and Tensor-Flow. Upon getting the result, we have compared the smell value and density obtained from both the software (Tensor-flow developed by the professionals and QGIS developed by the researchers). This approach helped us in validating our measurement and analyze the rate of increase in code smell in both software.

we have divided our approach into two categories: 1. Research Approach and 2. Code Smell Measurement.

3.1 Research Approach

For our study, we have selected the 10 version of QGIS (one from every year starting from 2013) and 5 version of Tensor-flow. We had use tool to UNDERSTAND (a software static analysis tool) on all the version to extract the basic parameters like LOC, the number of classes and methods, commented lines, coupling between various classes, modules and then use those parameters in our formula to quantify the code smell density and value to study the evolution.

In a similar way, we have used PMD-CPD is used to detect the code clones in the below-mentioned releases. The main reason for selecting the QGIS and TensorFlow codebases is because of similarity in programming languages, their codebase size and high star rating on

GitHub. 10 releases of QGIS namely (Rel 1.0 (year-2009), Rel 1.8 (year-2012), Rel 2.0 (Year-2013), Rel 2.6 (Year-2014), Rel 2.10 (Year-2015), Rel 2.12 (Year-2015), Rel 2.16(Year-2016), Rel 2.18 (Year-2016), Rel 3.0 (Year-2018), Rel 3.2 (Year-2018)) and 5 releases of Tensor-Flow namely (Rel-1.1, Rel-1.2, Rel-1.3, Rel-1.5, Rel-1.7).

We have device smell measurement formula for Large class, Large Method, Lazy Class, Lazy Method, Large comment to code Ratio, Small Comment to code ratio which we are going to talk in detail.

We have applied our quantification on 10 releases of QGIS and 5 Releases of Tensor-flow to study the severity of code smell and their evolution.

Below is the smell measurement formula for all the code smell along with the threshold value and selection criteria which we have used in our experimentation.

3.2 Large Class

A Class is considered to be a large class when it is trying to do too much, contains many variables and methods. To sum up, it should contain a large amount of code in a class. Classes tend to grow over time with the new version update. It is always easier to understand the class with clean and minimal code. The solution to large class smell is to use proper refactoring techniques to divide the large class into two or more class of less complexity, introducing such changes early in the software development process could save a lot of money and time. In our analysis, we have considered 750 lines as a threshold value. We have used tool to understand to extract the basic information from both software's (QGIS and Tensor-Flow) by generating the metrics report. Phongphan Danphitsanuphan and Thanitta Suwantada [28] used the predefined rules set for the detecting the bad smells. Software metric rules [28][29] were used for setting the

threshold value for large class code smell. The report comprises of many details such as total code length, commented lines, coupling factor, blank lines, cohesion ratio, cyclomatic complexity and many other factors for all Classes, methods, and packages present in the software.

There were 15 reports (one report per release) are generated by the help of static analysis tool "UNDERSTAND". The generated reports are very detailed and contain many data points on the software, so we have filtered out the content as per our requirement. The filtering criteria are to include all the classes (Selected criteria - Abstract, public, private, class and exclude the rest) detected by the static analysis tool. Once all the classes are sorted, we have used an exponential function to calculate the code smell density and value for all the releases of QGIS [33] and Tensor flow [34].

The exponential function [lc] check for an extent to which the code length of a class is far from the threshold value (which is 750 lines). The class which is far away from the threshold value is penalized more than class near to the threshold value. (Ci – Cth) returns the distance of a class from the threshold. Using this approach, we ran a loop on all the classes of QGIS and Tensor-flow and perform the summation of all the smell value obtain from (Ci – Cth) to get the overall code smell value. Upon retrieving the Overall smell value, we then take a square root in order to normalize the result.

Required: Length of code (LOC) for each class

Large Class Code Smell Value $\boldsymbol{V[lc] = \sqrt{\sum(Ci - Cth)\text{^}2}}$

Where Ci = Length of Code for a single class

Cth = Threshold value

Large Class Code Smell density $\boldsymbol{D[lc] = \sqrt{\sum(Ci - Cth)\text{^}2}}$ **/ Total LOC of all classes**

Selection is made where Class LOC is greater than 750 lines

Overall large class code smell value depends upon the number of the class present in opensource software. The smell value may rise if the classes are more. That is the reason, it is also important to know the code smell density of an open source software by dividing Code smell value by total Length of the code of all classes to get the correct approximation.

3.3 Large Method

The method is termed as a large method if it possesses a huge amount of code and trying to perform many operations at once. Due to this, it is not easy to understand the exact role of a function. The solution is to use the proper refactoring methods to break the function in two or more than two depending on its complexity. Most of the organization have the proper coding standard document where they mention the expected length of a function. We have considered 50 lines as a threshold value for the large method. Similar to large class code smell, we have extracted the 15 reports on the above-mentioned version of QGIS and Tensor-flow by using the static analysis tool understand. We have selected the methods which are beyond 50 lines of code. Software metric rules [25][26][27] were used for setting the threshold value for large method. The filtering criteria is to include all the methods (Inclusion Criteria – public function, private function, function, private const function, public const function, private static function, private function, protected function, protected virtual function, protected const function, static function, protected virtual const function, public virtual function, public virtual const function / Exclusion Criteria - rest) detected by the static analysis tool. Once all the methods are sorted, we have used an exponential function to calculate the code smell density and value for all the releases of QGIS [33] and Tensor flow [34]. Having a large method increases the complexity and impact the performance of overall software. For an example, there is method “X” which gets (1) time and

(2) date and it is called by many classes out of which few classes need time and few classes need the date. In both cases, there is a set of code in method "X" which is not executed but still, it is been read by compiler. If this method "X" is called a thousand times during daily execution, then there is an obvious performance impact.

We applied an exponential function on all methods/function present in the 15 releases of QGIS and Tensor-Flow. An exponential function calculates the difference between the threshold value and the code length of a method. This helps us to identify the methods which are beyond the threshold value and penalize them as per their distance.

Required: Length of code (LOC) for each Method

Large Method Code Smell Value $\boldsymbol{v[lm]} = \sqrt{\sum(\boldsymbol{Ci} - \boldsymbol{Cth})\text{^}\boldsymbol{2}}$

Where Ci = Length of Code in a single Method

Cth = Threshold value

Large Method Code Smell Density $\boldsymbol{D[lm]} = \sqrt{\sum(\boldsymbol{Ci} - \boldsymbol{Cth})\text{^}\boldsymbol{2}}$ **/ Total LOC of all Method**

Selection is made where method LOC is greater than 50 lines

Similarly, large method code smell density (D[*lm*]) is calculated by dividing the smell value by the total code length of all method.

3.4 Lazy Class

The class qualifies for a lazy class if possess less amount of code and literally doing nothing. The better solution for such issue is to merge the class in other. It is important to maintain the code in long run. After a certain point, decreasing the class length can impact readability. We used a function ($\boldsymbol{CTh} - \boldsymbol{Ci}$) which checks the distance between threshold value and class code length. The greater the difference between class length and threshold value, the more we are going to penalize it.

Required: Length of code (LOC) for each class

Small Class Code Smell Value $\boldsymbol{v[sc]} = \sqrt{\sum(\boldsymbol{CTh} - \boldsymbol{Ci})\text{^}\boldsymbol{2}}$

Where Ci = Length of Code for a single class

Cth = Threshold value

Small Class Code Smell Density $\boldsymbol{D[sc]} = \sqrt{\sum(\boldsymbol{CTh} - \boldsymbol{Ci})\text{^}\boldsymbol{2}}$ **/ Total LOC of all classes**

Selection is made where Class LOC is less than 100 lines

Similar to the above-mentioned code smell, we applied this function on 15 releases of QGIS and Tensor-flow. Code length for each class is needed to apply our measurement, we use "UNDERSTAND" to get all required details. We are also calculating the Small class smell density by dividing the overall total code length of all classes. Inclusion and exclusion criteria are the same as used in large class code smell. Sukhdeep Kaur and Dr. Maini [24] talks about the detection of lazy class in his paper "Analysis of various software metrics used in detecting the various bad smell" but they didn't talk about quantification and severity of code smell. We have taken the threshold value for the small class from [24] which is 100. We have selected the classes which are below threshold value for our analysis. Result of our analysis is provided in the next chapter.

3.5 Lazy Method

A method is said to be lazy when it possesses few lines of code and such small method can be easily incorporated in some other class or methods. For our analysis, we have selected 5 lines as the threshold value [30]. Methods which less than 5 lines of code are selected from all 15 releases of QGIS and TensorFlow. Most of the researcher today do not worry about the short method because the compiler has evolved over time. Increased number of function impact is too negligible that no one worries about it but a function with than 5 lines of code does not even qualify as the function. The method with 5 lines of code does not impact the performance of the

program and hence it can be merged with other classes or method. By doing so, we can normalize the number of methods present in the software.

Similar to small class code smell, we have used an exponential function on all methods and function of opensource software to get the code smell value and density. We have selected all the methods which has less 5 lines of the code and the exponential function ($\boldsymbol{CTh - Ci}$) checks distance of method from the assigned threshold value. Inclusion and exclusion for selecting the methods are same as mentioned in the large method code smell.

Required: Length of code (LOC) for each Method

Small Method Code Smell Value $\boldsymbol{v[sm]} = \sqrt{\sum(\boldsymbol{CTh - Ci})^2}$

Where Ci = Length of Code in a single Method

Cth = Threshold value

Small Method Code Smell Density $\boldsymbol{D[sm]} = \sqrt{\sum(\boldsymbol{CTh - Ci})\text{^}2}$ **/ Total LOC of all method**

Selection is made where method LOC is less than 5 lines

We then use the square root to normalize the value of our measurement.

3.6 Large Comment to Code Ratio

Code commenting is always been considered as the best coding practice. We certainly use the comment for our peer programmers to understand the newly written or existing code. It even helps a non-technical manager to understand the program without learning the programming language. It helps 2 programmers who were making changes in the same program at the same time. Code commenting certainly has lots of benefits, but excessive commenting could be a sign of a code smell. Few people have started making design and patterns while commenting on the code, we should all understand this commenting is an art, but it is not an art

project. It is not good to comment at every line that just means that code is very complex and that's the reason to explain it at every point. Excessive commenting could impact readability.

As per the Infospheres team of Caltech has this to say about the comment-to-code ratio in their coding standard document ("http://www.sourceformat.com/coding-standard-java-caltech.htm#Documentation") : A code to comment ratio of 25% is termed as the excellent commented code. For program submission a code with comment ratio less than 50% is expected not more than that. Documentation ratio of more than 50% in code is not considered.

For our analysis, if the code to comment ratio is more than 0.5, then we are considering it to be a bad code smell. We have used a function which calculates the distance between the commented lines and the threshold value (**0.5 * length of the code in a Class**). Using this approach, we get to know the severity of the code smell. A Class with a more distance gets more penalty than a class with less distance.

We have used this function on the classes which code to comment ratio more than 0.5. Inclusion and exclusion criteria for selecting the classes are the same as mentioned in the large class code smell.

Required: Total Length of code (LOC) for all class

Large Comment to Code Ratio Smell value $\boldsymbol{v[lcr] = \sum(\text{Commented lines} - (0.5 * loc))}$

Large Comment to Code Ratio Smell Density $\boldsymbol{D[lcr] = \sum(\text{Commented lines} - (0.5 * loc))}$**/ Total No of classes**

Selection of classes is made where code to comment ratio is greater than 0.5

15 reports have created one for each version of QGIS and Tensor-Flow by using the static analysis tool. We sorted all the classes whose code to comment ratio is more than 0.5 and then apply the code smell quantification on them. Along with code smell value, smell density is also calculated to get the correct approximation.

3.7 Small Comment to Code Ratio

Developers often spend much time to understand the code while making a change in a new program. When you are building new functionality in the same product then code commenting could have been a way to make the task much easier. There are many data units in a program where we can keep the comments like variable initialization, input and output parameter reference, return value and defining the function. The comment has to precise as it should be clearly understood by the peer programmers. if your code is too complex to understand, then your code is just bad. On the other hand, less commented program to hard for non-technical managers, technical writers who document the functionality.

For this particular code smell, we have selected the classes which is less than 0.1 code to comment ratio of QGIS and Tensor-Flow. As per the Infospheres team of Caltech has this to say about the comment-to-code ratio in their coding standard document ("http://www.sourceformat.com/coding-standard-java-caltech.htm#Documentation") : That code is termed as poorly commented if comment ratio counter falls between a range of 5% - 10 %. Considering that approach, we have used a function which calculates the distance between the commented lines and the threshold value (**0.1 * length of the code in a Class**). Using this approach, we get to know the severity of the code smell. A Class with a more distance gets more penalty than a class with less distance.

Required: Total Length of code (LOC) for all class

Small Comment to Code Ratio Smell Density $\boldsymbol{v[scr] = \sum((0.1 * loc) - \text{Commented lines})}$

Small Comment to Code Ratio Smell Density $\boldsymbol{D[scr] = \sum((0.1 * loc) - \text{Commented lines})}$ **/ Total No of classes**

Selection of classes is made where code to comment ratio is less than 0.1

Inclusion and exclusion criteria for selecting the classes are similar to the large class code smell. We have extracted reports for all version of QGIS and Tensor-flow. Once we have report metrics, we have sorted the class which is below 0.1 code to comment ratio and then using them for our analysis. We have discussed the result in the next chapter.

3.8 Codes Clones

There are many parameters that are useful to judge the quality of the code. Code clones are one of them. Duplicate code often refers to a sequence of code occurs more than once across multiple programs. Duplicate code issue occurs when multiple developers make changes in the same codebase or sometimes to meet the deadline, the novice developer may not be able to resist themselves from the temptation of performing copy and paste. There is another form of duplication as well when two parts look different, but they perform the exact same task. The solution of the duplication issue is to find the most copied code and then make it more generic function/include-file so that it can be used over and over again without actually copying it. By doing this, the size of the codebase could be reduced to an extent.

For our analysis, we have used PMD-CPD (code-clone detection software) to detect the code clones from both opensource software QGIS and Tensor-Flow. We have used the standard threshold limit of 75 set by the PMD-CPD tool. PMD-CPD tool considered to be a match if it detects 75 lines copied from another part of the same codebase. We have PMD tool to detect code clones in three languages which are C++, Python, and Objective – C. QGIS and tensor-flow is majorly written in these three languages and that's been the reason of selecting it.

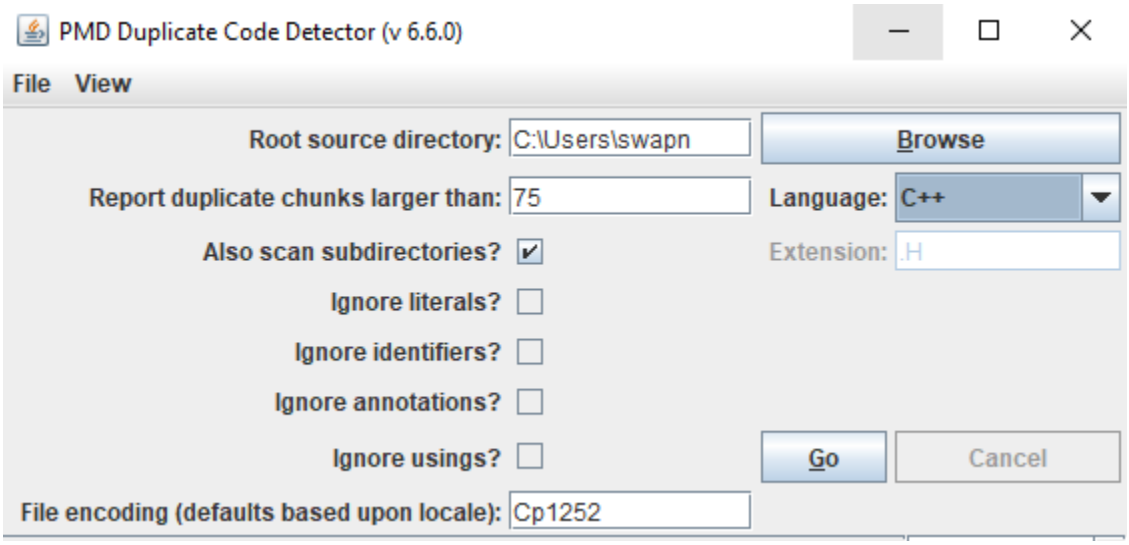


Fig: 3.1 – Snapshot from Tool PMD-CPD detecting C++ Code Clones

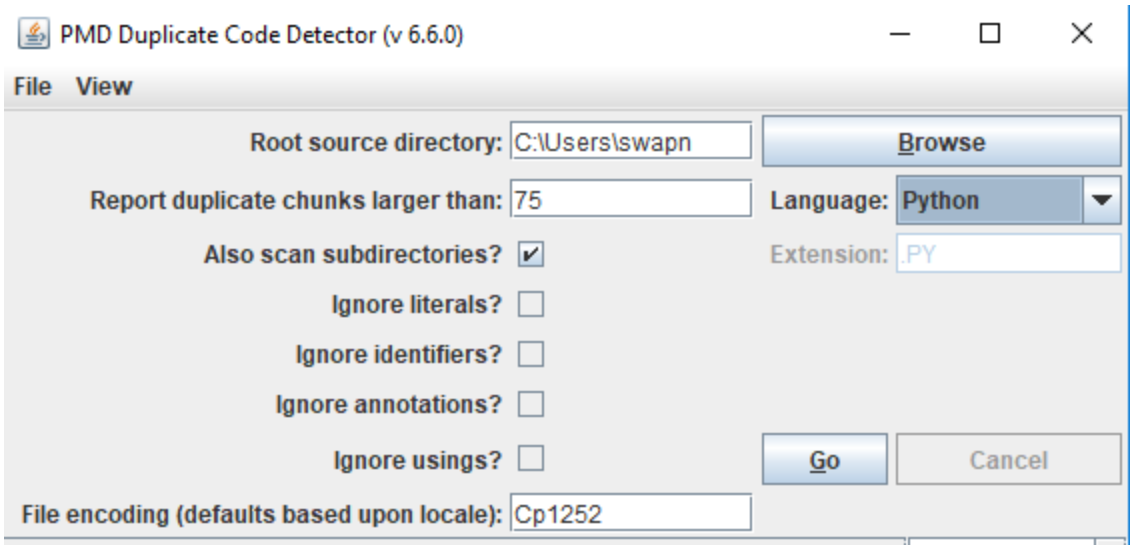


Fig: 3.2 – Snapshot from PMD-CPD detecting Python Code Clones

we have generated the report for 10 releases of QGIS namely (Rel 1.0 (year-2009), Rel 1.8 (year-2012), Rel 2.0 (Year-2013), Rel 2.6 (Year-2014), Rel 2.10 (Year-2015), Rel 2.12 (Year-2015), Rel 2.16(Year-2016), Rel 2.18 (Year-2016), Rel 3.0 (Year-2018), Rel 3.2 (Year-2018)) and 5 releases of Tensor-Flow namely (Rel-1.1, Rel-1.2, Rel-1.3, Rel-1.5, Rel-1.7). Overall 45 reports (one report per language) were generated from the 15 version of QGIS and Tensor-Flow. Result of duplicate code smell is provided in next chapter.

CHAPTER 4: RESULT AND KEY FINDINGS

In this Chapter, we are going to talk about our research design, results which we have obtained by applying our quantification measure on opensource software. In the end, a comparative analysis was performed.

4.1 Results

In this study, we analyze seven code smells as a metric for code quality. For each code smell, we identify a threshold beyond which a code unit is tagged with the smell. We calculate two measures; smell value calculated by measuring the distance a code unit from the smell, code smell density is calculated by dividing the total smell value by the number of analyzed lines of code.

For our analysis, we ran our quantification measure on 10 releases of QGIS and 5 releases of Tensor-flow and calculated bad smell value and density. We have stated the result of our experiment in this section.

In this Section, we are going to talk about code smell density and value obtained after applying our quantification measure.

4.1.1 Code smell Density and Value

Large Class

The threshold value which we have considered for the Large class code smell is 750 lines. Any class beyond the threshold value is considered as Large class code smell. For our analysis, we first generated 15 metrics reports using static analysis tool “understand”, then identified the classes which are exceeding the threshold value and then, in the end, we applied our measurement on those specific classes. Smell density is calculated by dividing the smell value by total LOC of all classes.

In a very early release Rel 1.0, 41 classes were qualified for the large class while in the latest release 3.2, the number of large class increased to 258. The highest smell density of 7.4 was recorded in the early release Rel 1.0 and the latest release has shown the density of 2.9.

Overall, it does not impact the smell density since the codebase size is also increasing with every release. The Smell density of QGIS shown an obvious decreasing trend. On the other hand, Large classes detected in all the releases of tensor flow [34] fall under the range of 150. The number of the large class has decreased in the latest releases Rel 1.7 from 139 to 112.

The number of large classes detected in Tensor-Flow is far less than the classes recorded in QGIS and over time it is reduced, while on the other hand the number of large classes in QGIS is increased with every coming release.

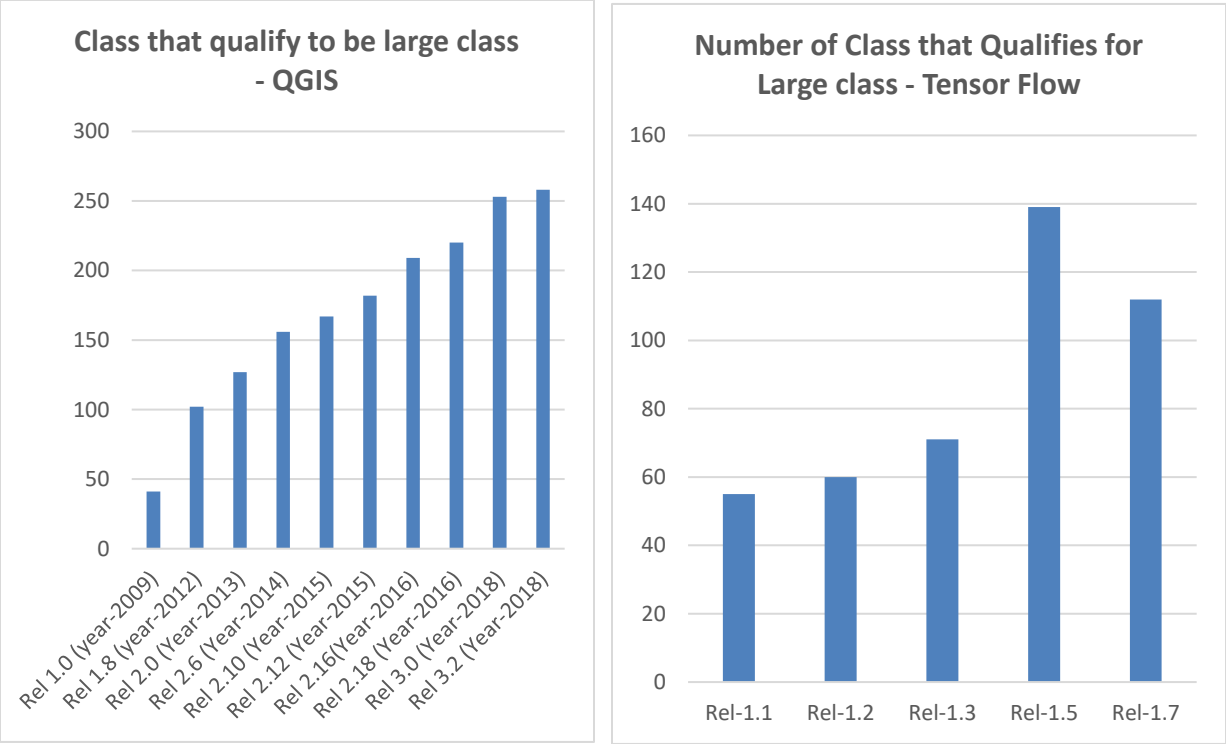


Fig: 4.1 Number of large classes in an opensource software

The Smell density of 1.5 has recorded in the early release Rel 1.1 of tensor flow and it keeps on reducing with a new release. The obvious reason being the code size of tensor flow has increased with every release but smell density is way less compared to smell density of QGIS. The declining smell trend in tensor flow and QGIS represent the proper usage of refactoring method.

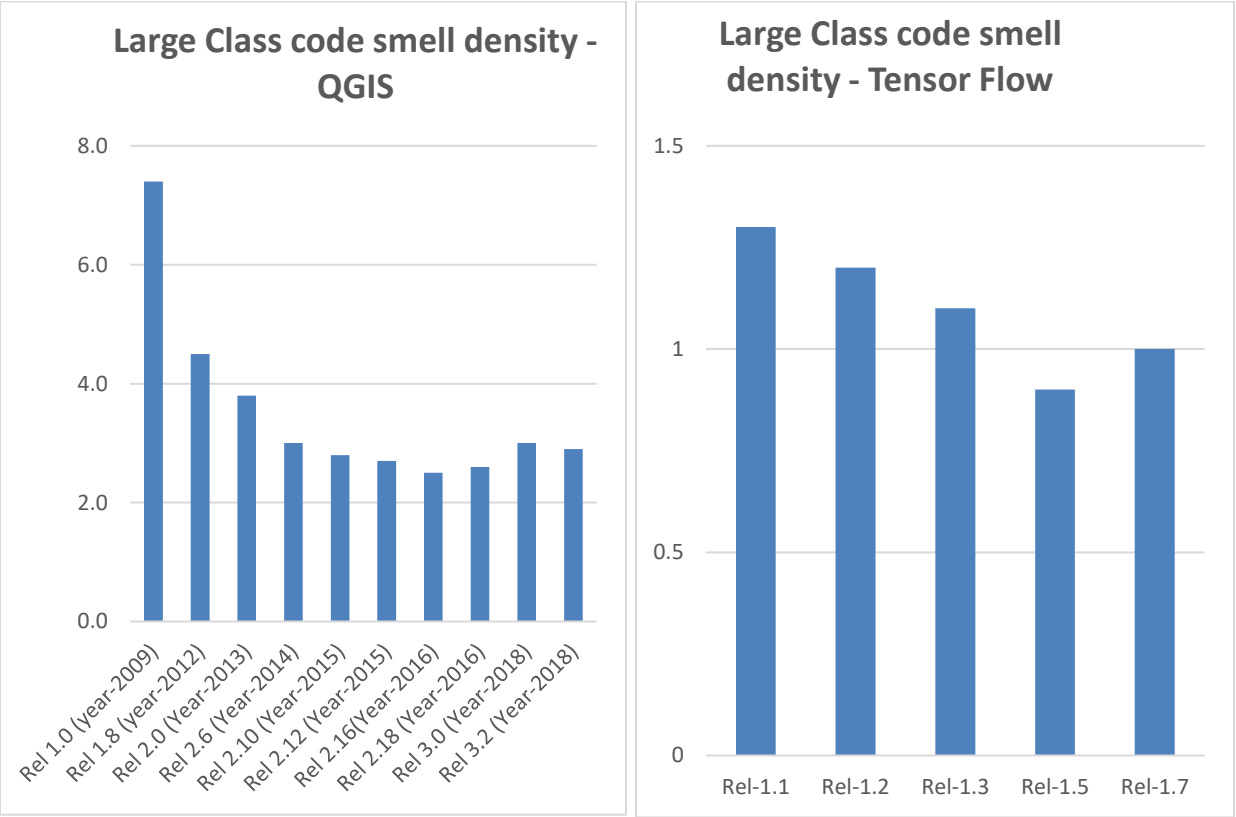


Fig: 4.2 Large class smell density in an opensource software

On the other hand, code smell value in QGIS is increasing with the next release, this provides us a clear indication that the number of large classes has increased very rapidly. The highest jump in smell value can be observed in the Rel:3.0 from Rel: 2.8. The reason for this event is the highest increase in a large class is recorded in between Rel 2.18 and Rel 3.0. The smell value

observed in the QGIS is way higher than in tensor-flow. The highest smell value recorded in tensor-flow is 8191.80 in Rel 1.5 and from there, it starts depreciating.

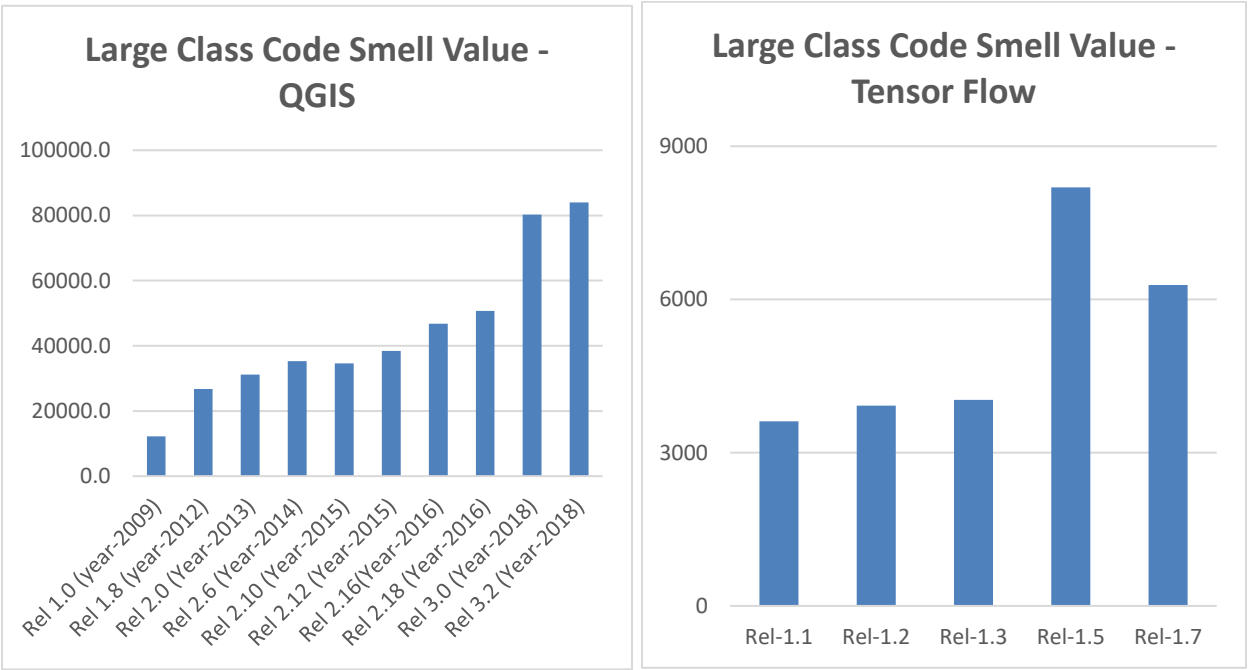


Fig: 4.3 Large class smell value in an opensource software

When we compare Smell density, smell value of QGIS and Tensor-Flow, we have observed professionally developed software (Tensor-Flow) possess less smell density and value than a software developed by reseachers (QGIS).

Large Method

The threshold value which we have considered for analyzing long method code smell is 50 lines of code that mean if a method possesses more than 50 lines of code then it is classified as a long method. We have applied our quantification measure on the methods obtain from various releases of QGIS and Tensor flow. we have generated 15 reports from the releases of QGIS and

Tensor Flow using static analysis tool “Understand”. From the report, we have extracted the function list which possesses more than 50 lines of code and then applied the formula on those methods to measure the smell value and density.

We have calculated smell value and density for all 15 releases of QGIS and Tensor-flow to understand the smell evolution in opensource software.

As we can see fig 5.4, the highest QGIS smell density of 4.0 is recorded in Rel 1.8. From Rel 1.8, it starts depreciating and almost become constant until the latest release. Similarly, the QGIS long method smell value decline after the very third release. The lowest density of 0.7 were recorded in Rel 2.16. This trend clearly indicates the refactoring methods is put into use in order to lower the complexity of the methods.

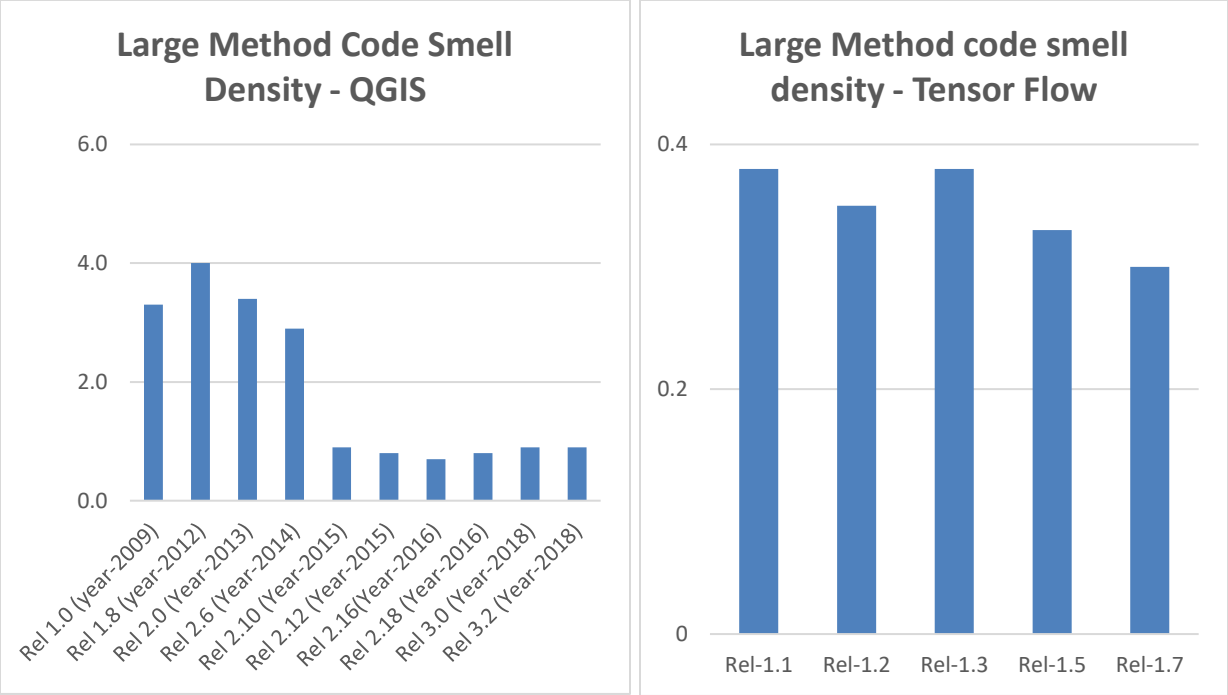


Fig: 4.4 Large Method smell density in an opensource software

The long method of smell density in tensor flow way low as compared to QGIS. The highest smell density of 0.38 was recorded in Rel 1.1 and Rel 1.3 and it is been very consistent throughout the releases by maintaining the smell density of 0.3. In the latest release Rel 1.7, the smell density depreciates by 0.05. This indicates that the tensor flow has been built by keeping large method smell into consideration. we have observed that the long method density of Tensor-Flow is way less than the density of QGIS.

There is the large number of long methods recorded in the QGIS than in Tensor-Flow, that is clearer from the smell value. As you can see in fig 5.5 data trend, the code smell value for tensor-flow falls under 25000 while the lowest recorded long method smell value of QGIS is 27640 and highest smell value goes till 498051.6.

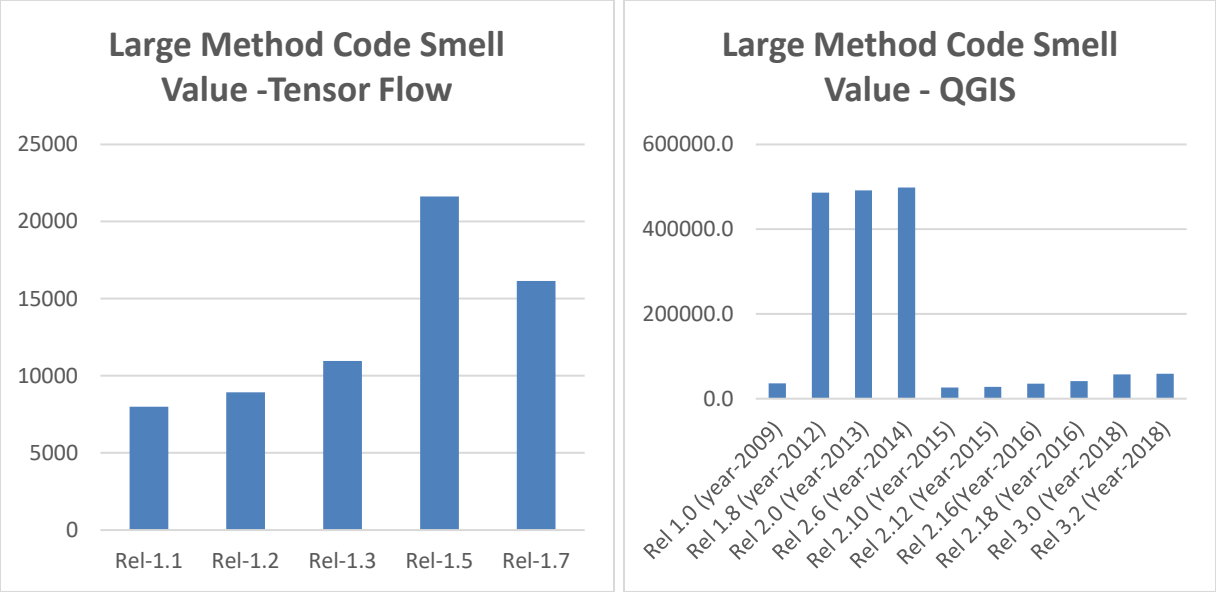


Fig: 4.5 Large Method smell value in an opensource software

Method should be smaller as possible, LOC is the indicator of the bad code. Thus, by reducing the complexity overall performance of a program can be improved.

Lazy Class

The Threshold value selected for the Lazy class is 100 lines of code that mean if a class possess a code less than 100 lines than it is qualified to be a lazy class. To study the evolution of this code smell, we had used static analysis tool "Understand" on 15 releases of QGIS and Tensor-flow and created the report listing the classes which are below 100 lines of code for all the releases. Once we have the report, we have applied our quantification measure, we have calculated the lazy class value and density of all release of an opensource software.

As we can observe in the Fig: 5.6 data trend, lazy class smell density in QGIS is declining in latest release Rel 3.0 and 3.2 after maintaining consistency in 7 consecutive releases starting from Rel 1.8 to Rel 2.18. The highest and lowest smell density recorded in QGIS is 0.62 and 0.2 respectively. We have recorded the smell density of 0.4 for 1.8 (year-2012), Rel 2.0 (Year-2013), Rel 2.6 (Year-2014), Rel 2.10 (Year-2015), Rel 2.12 (Year-2015), Rel 2.16(Year-2016), Rel 2.18 (Year-2016). On the other smell density recorded in Tensor-Flow is showing a declining trend. We have observed the smell density of 0.8 in very early release Rel 1.1 which is now in the latest release is reduced to 0.5. The main reason for density declines in both the opensource software is due to an overall increase in the code base size.

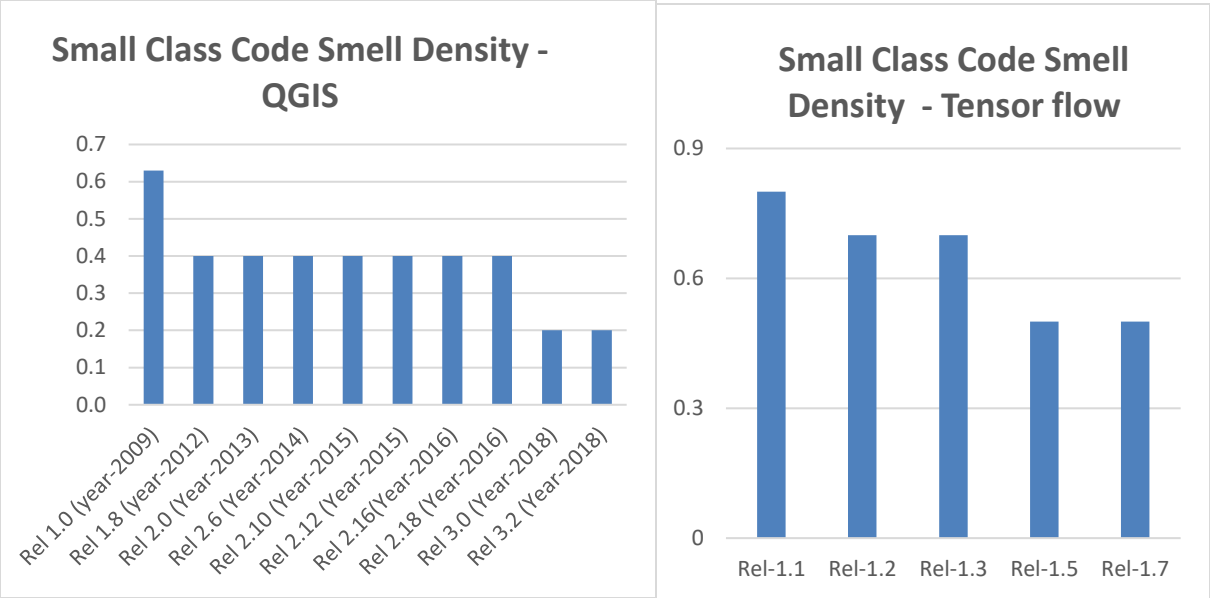


Fig: 4.6 Lazy Class smell density in an opensource software

A clear look at smell value trends in Fig: 5.7 indicates that there is an increase in the number of lazy classes in both the releases of QGIS and Tensor – Flow. when we talk in terms of QGIS, we have observed that smell value has increased very sharply in Rel 2.16 and then degraded. The smell value of 882.6 was recorded in early release Rel 1.0 and from there it has been increased to 16250 (Rel 2.16). Even, the smell recorded in latest release Rel 3.2 is 7082.8 which is again very low compared to the smell value of Tensor flow. The highest and lowest smell value of tensor-flow is 14173.33 and 32648.8 which were recorded in Rel 1.7 and Rel 1.1. There is a definite increase observed in the smell value of Tensor-flow due to this reason, we can say that the number of the lazy class in Tensor flow has increased with a sharp rate.

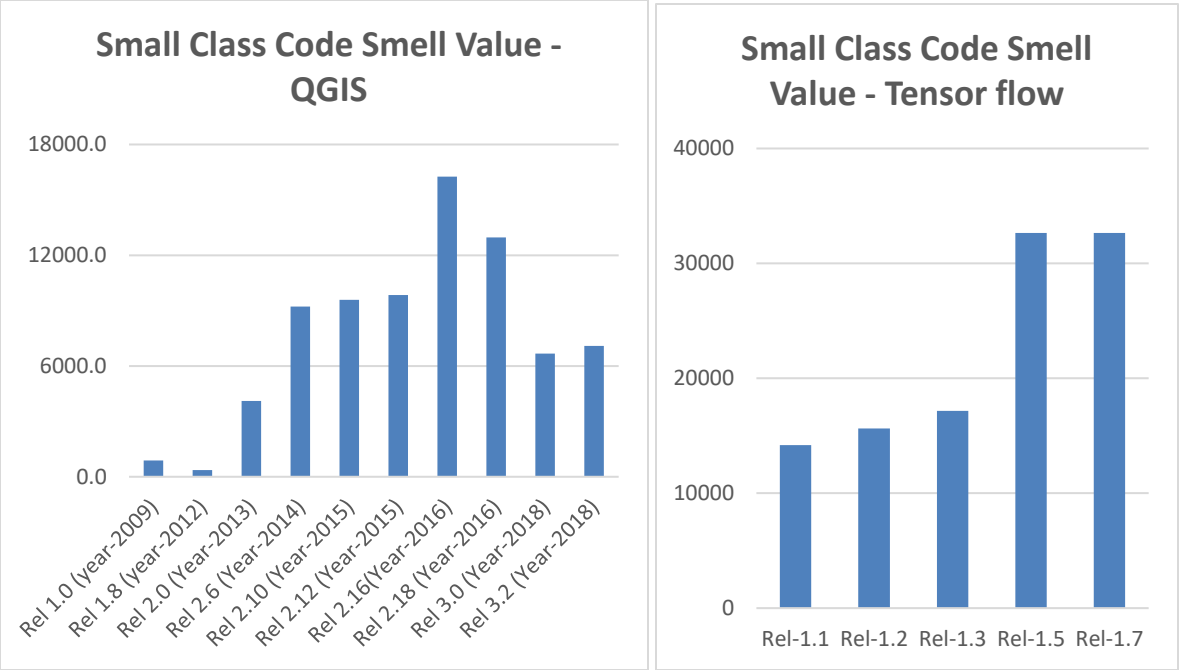


Fig: 4.7 Lazy class smell Value in an opensource software

Lazy Method

The threshold value selected for the large is 5 lines of code. A method qualifies for a lazy method if it has less than 5 lines of code. The static analysis tool is used to extract the list of all method present in the 15 releases of QGIS and Tensor-Flow. Upon getting the method list, we have filtered all method with 5 lines of code are below and created 15 files (one for each release). In order to analyze the smell evolution, we applied the quantification measure as stated in chapter 4 (4.4 lazy methods) on the filtered method list to obtain smell value and density.

As you see in Fig: 5.8, the smell density is pretty consistent in case of QGIS. The highest smell density of 0.6 is recorded in Rel 2.12. The average smell density of QGIS across all the releases is 0.44. On the other hand, the average density of Tensor-flow across its all releases is 1.7. The smell density in tensor flow has shown little hike and the average density is more than the smell density in QGIS. The highest recorded density in Tensor- Flow is in latest release Rel 1.7.

The smell density graph of an opensource software does not provide a clear idea on the number of lazy methods that have been added to releases. It is quite evident by the smell value trend that there is an increase in the lazy method in both the software.

In QGIS, the smell value increased from 1.5 in Rel 1.0 to 18.4 in Rel 3.2 while in Tensor-Flow the smell value increased from 17.03 in Rel 1.1 to 2580.3 in Rel 1.7.

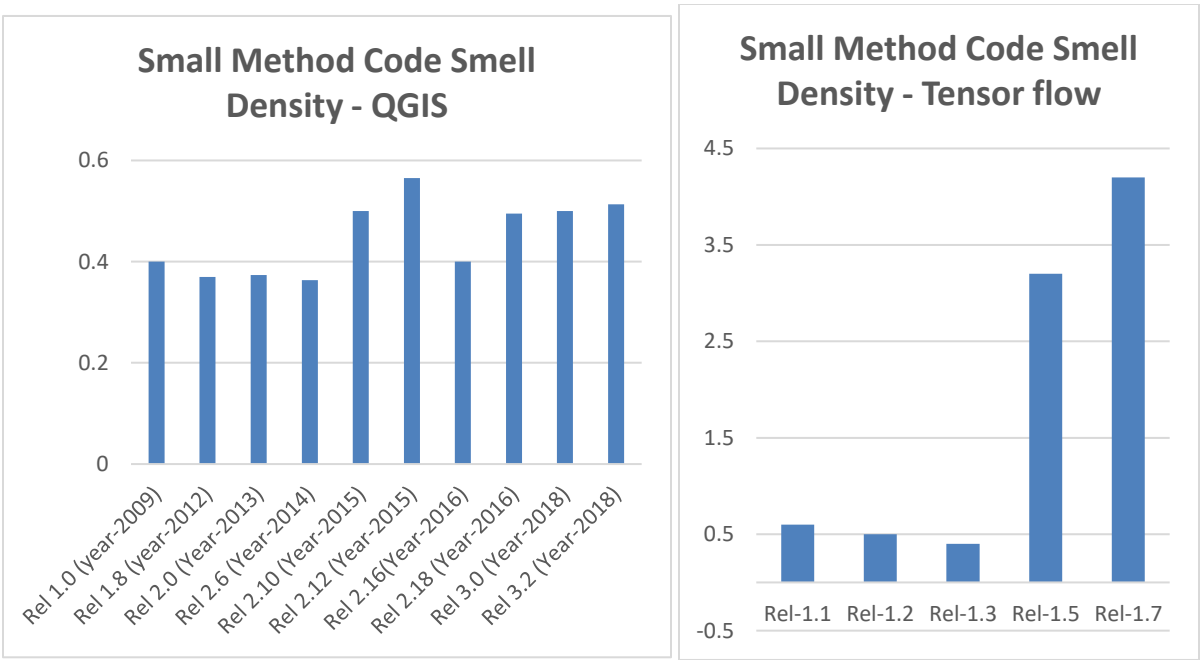


Fig: 4.8 Lazy Method smell density in an opensource software

The hike of 2562 in the smell Value is been observed in the last two year of tensor-flow while in QGIS, there is an increasing trend, but it is way lesser than that of Tensor-flow. The data for the smell value is given in Fig: 5.9 below for the reference.

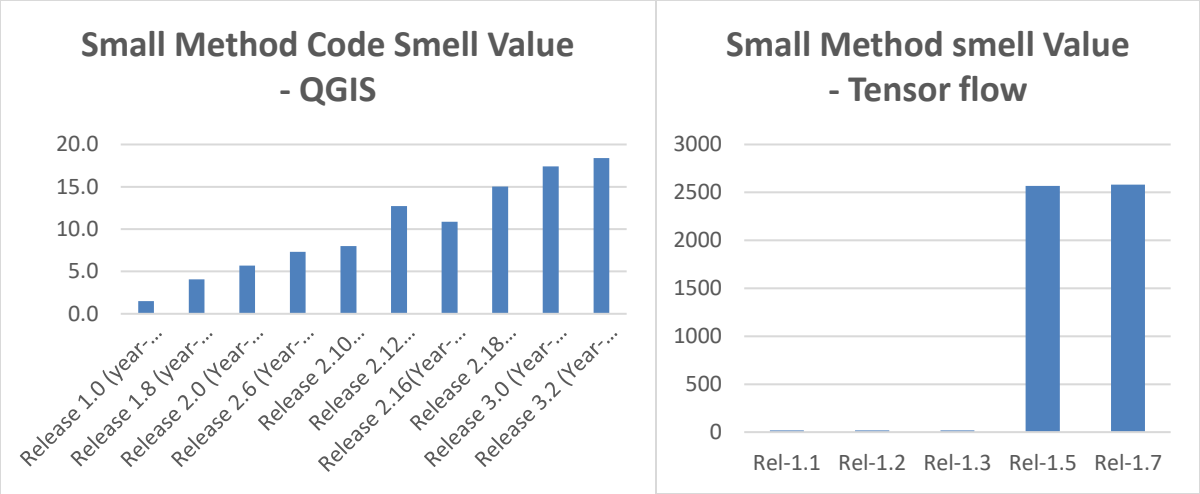


Fig: 4.9 Lazy Method smell Value in an opensource software

Large Comment to Code Ratio

It is always good to have documentation in programs, this would help non-technical managers to understand the code without any prior programming knowledge. Sometimes, over commented program impacts the readability of the code, or that just mean the code is too complex to understand. The threshold value which we have considered for this particular type of smell is 0.5 that means if the comment ratio for any class is greater than 50% of its overall code then that class qualifies for large code to comment ratio smell. For studying the evolution of this particular smell, we have 15 releases of QGIS and Tensor-Flow as our use case. We have used static analysis tool "Understand" to extract all the classes along with their total length of code and commented lines. We then applied check formula [$\sum(\mathbf{Commented\ lines} - (\mathbf{0.5} * \boldsymbol{loc}))$] on all the classes to know if a class possess this smell. Upon using the formula mentioned above, the classes which provide the positive result were qualifies for having this smell. We then filtered out those classes and used them further for our analysis. As a result, we have quantified the large comment ratio smell for all mentioned releases of an opensource software.

Fig: 5.10 represent smell density result which we have calculated from the releases of an opensource

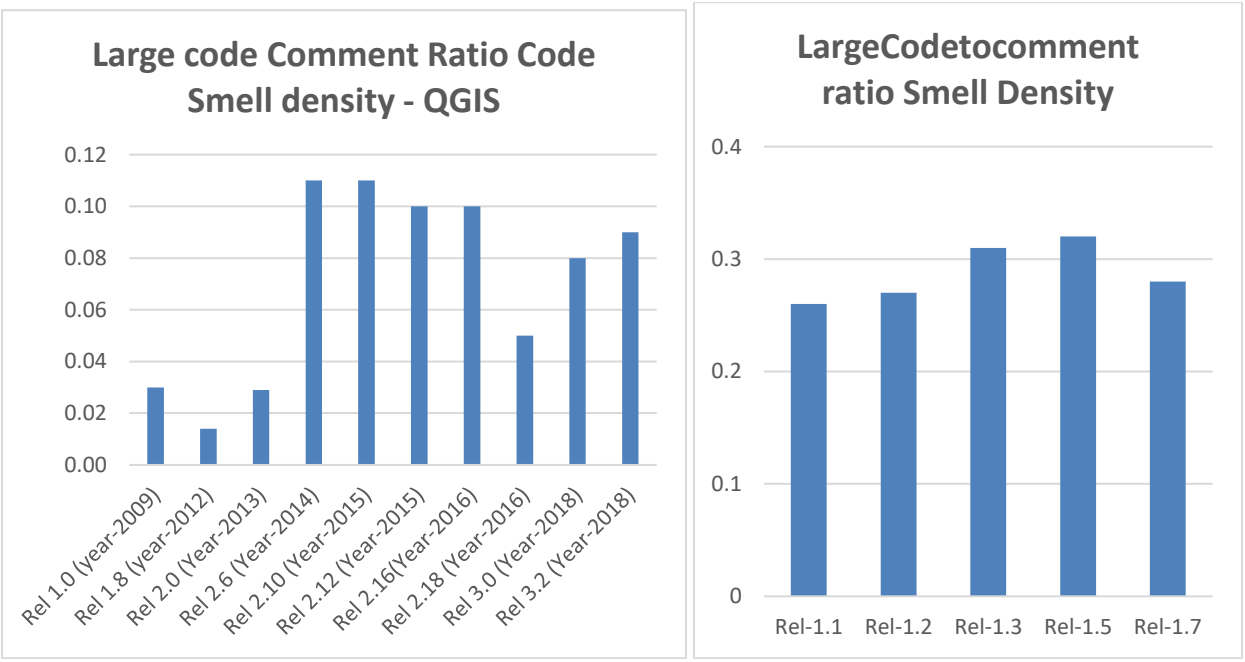


Fig: 4.10 Large code to comment ratio smell Density in an opensource software

software. As you can see figure above, the lowest smell density is observed in the early release "Rel 1.8" and the highest smell density recorded in QGIS is 0.11 (Rel 2.6), from there on it gets reduced to 0.09 which is way less compared to the density recorded in the Tensor – Flow. In Tensor – Flow, the highest density of 0.32 is recorded in Rel 1.5 and from there on it get reduced to 0.28. The smell density represents the ratio of smell present in the releases, but it doesn't talk about the severity of smell detected.

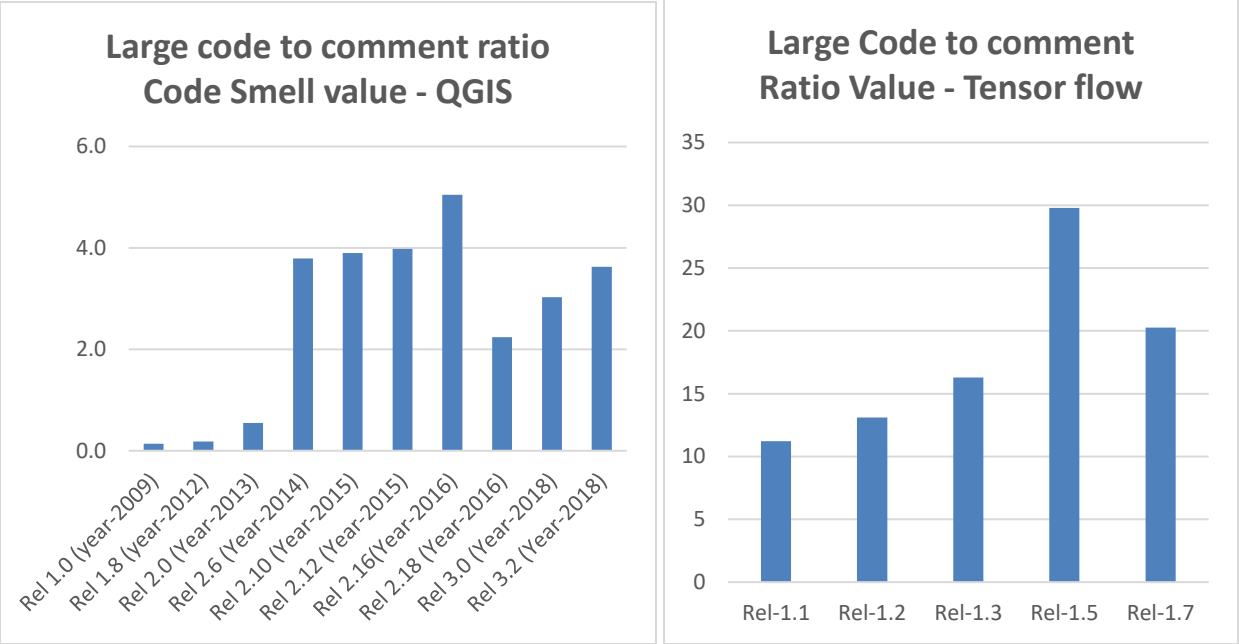


Fig: 4.11 Large code to comment ratio smell Value in an opensource software

We get to know the severity of smell by looking into the smell value trend, As we can see in Fig: 5.11, it is clearly indicated that Tensor – Flow has large smell value when compared to the smell value of QGIS. There are a large number of classes which qualifies for large comment ratio in the Tensor flow.

Small Comment to Code Ratio

Comments are meant to communicate the idea behind the code. Poorly commented code cause impediment in understanding the program, it becomes very difficult when two or three programmers are making the changes in the same code. The threshold value which we have selected for small comment ratio smell is 0.1 that means if the comment ratio for any class is less than 10% of its overall code then that class qualifies for the small code to comment ratio smell. For studying the evolution of this particular smell, we have 15 releases of QGIS and Tensor-Flow

as our use case. We have used static analysis tool "Understand" to extract all the classes along with their total length of the code and commented lines. We then applied check formula $[((\mathbf{0.1} * \boldsymbol{loc}) - \mathbf{Commented\ lines})]$ on all the classes to know if a class possess this smell. Upon using the formula mentioned above, the classes which provide the positive result qualified for having small comment ratio smell. As similar to the Large Comment ratio smell, we have extracted the smelly classes and used them to calculate the smell density and value. Total 15 reports (one for each release) were prepared and used to study the smell evolution throughout the software releases.

As you can see in fig: 5.12, the highest and lowest density recorded in QGIS is 5.4 and 3.3 respectively. smell density in QGIS is maintaining consistency and attain an average of 4.5. On the other hand, Tensor flow smell density in all releases lies between 2.5 and 3 maintaining an average of 2.75. Smell density in QGIS is more than the density recorded in Tensor-flow.

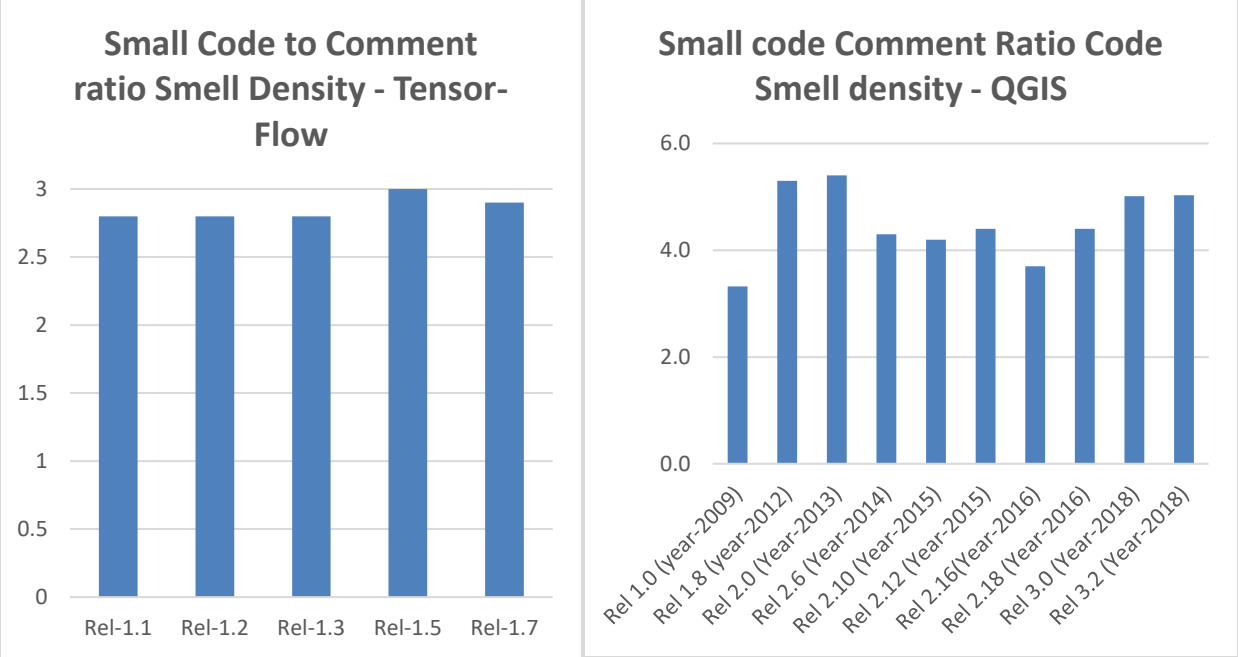


Fig: 4.12 Small code to comment ratio smell density in an opensource software

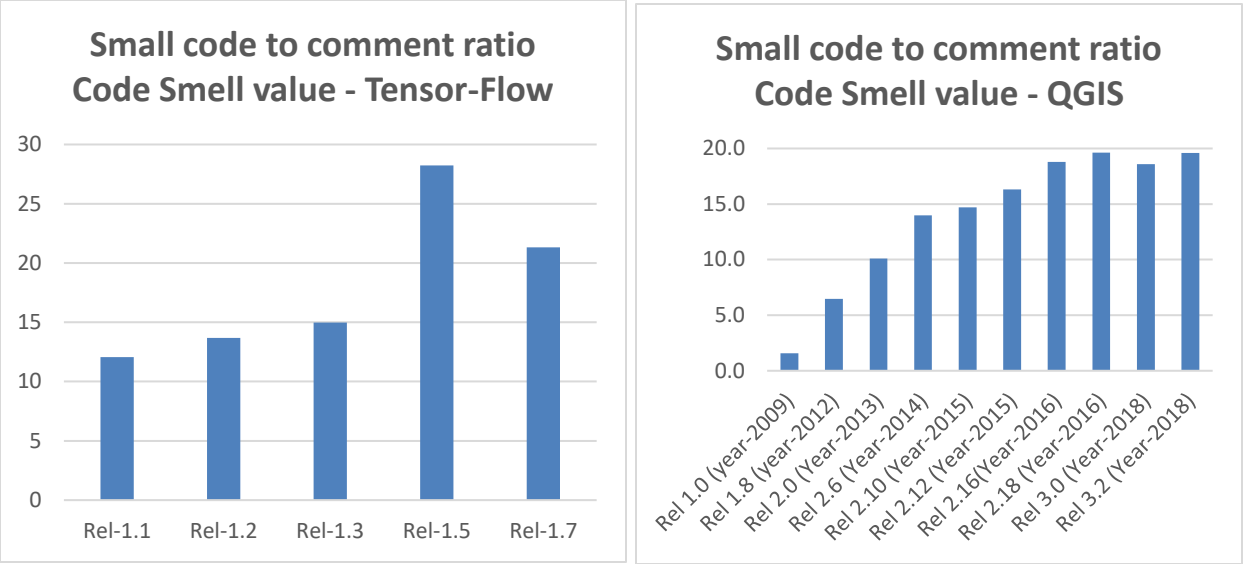


Fig: 4.13 Small code to comment ratio smell Value in an opensource software

The smell value in QGIS is getting increased with next release while Smell value in tensor flow has decreased by 9 points in the latest release.

Code Clones

PMD-CPD [32] has detected the significant number of copy-pasted code clones in all the version of evaluated software's. We have selected the standard threshold value of PMD-CPD [32] which 75 lines of code that means Clone detection tool would be considered a match if detects 75 lines copied – pasted from the parts of a software.

Clones In QGIS

Programming language C++ and Objective C constitute 70% of QGIS software. we have observed that early release "Rel 1.0" has 5.5% of overall Codebase possess duplicate codes and latest release "Rel 3.2" has 12% of total source code is copied-pasted in QGIS. There is an increase

of 7% if we compare the latest release from a very early release of QGIS. It could consider as a major performance impact and needed to be resolved by creating more generic methods or by increasing the code reusability.

It is been observed from the result that code clones in QGIS increases as the code size of opensource increases. Overall 410 code clones were reported in an early release [Rel 1.0 (Year-2009)] of QGIS and from there, it goes on increasing with every next release.

The highest number of duplicate code chunks which are 3305 was recorded in Rel 2.16 (Year-2016). C++ programming language tends to have more duplicate codes throughout the releases while on the other hand Objective C has few code clones.

The major programming language is C++ which 48% of entire QGIS and that's the reason for possessing the highest number of code clones. Python is followed by C++ and XML, it is 38% off entire QGIS and that's the reason, we have the highest number of duplicate codes in python after C++.

After Rel 2.16 (Year-2016) Overall code clones were reduced from 3305 to 2073 but as the codebase increases, the duplicate code starts growing till the latest release. The result from the last three releases namely Rel 2.18, Rel 3.0, Rel 3.2 shows that there is no change in the approach to write or to make the change in the programs.

Sometimes programmers try to replicate functionality from other modules. The bad thing about the copy-pasted code is about is that a programmer stops reusing the code. Data and trends are provided fig 5.2 and 5.3 below.

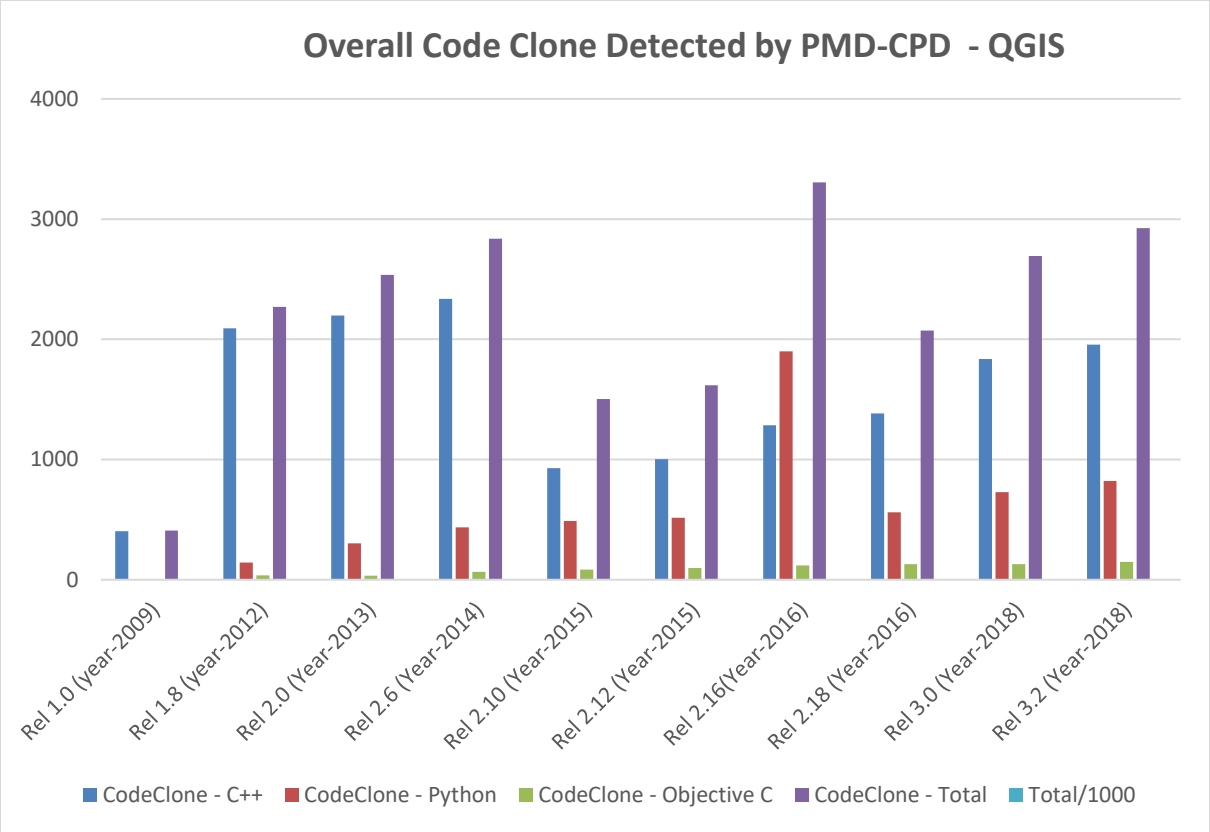


Fig: 4.14 Duplicate code detected by the PMD-CPD in QGIS

Table: 4.1 Number of code clones detected by the PMD-CPD in QGIS

QGIS Version	Code Clone - C++	Code Clone - Python	Code Clone - Objective C	Code Clone - Total	Code clone percentage
Rel 1.0	404	6	0	410	0.053310726
Rel 1.8	2092	142	36	2270	0.125758429
Rel 2.0	2197	304	35	2536	0.117081941
Rel 2.6	2337	435	65	2837	0.096288707
Rel 2.10	929	490	85	1504	0.054382171
Rel 2.12	1004	515	99	1618	0.05412019
Rel 2.16	1285	1900	120	3305	0.103100386
Rel 2.18	1384	560	129	2073	0.022218224
Rel 3.0	1835	728	129	2692	0.088093033
Rel 3.2	1955	822	149	2926	0.092446461

Clones in Tensor Flow

Similar to QGIS [33], Tensor flow [34] mainly comprises of C++, Python and Objective C. we have observed that there is 9.0% -- 11% of total source code is copied-pasted in Tensor flow, but we do not see any major increase when compare the latest release from the early releases of Tensor-Flow. There is an increase of 2% and trends show the value is going down with the latest release. The duplicate codes tend to increase as the size of Overall codebase. There is an obvious performance impact, but we have observed a code duplication decline in the latest release. The highest clones were recorded in Rel: 2.6 which is 11109 and from there on it reduced to 8487 in Rel:2.10. The clones detected in the Tensor Flow [34] are much higher when compared to the QGIS but then again, we also need to look at the Overall code size of both the opensource software. Python tends to have more duplicate codes than any of the programming languages across all the releases of Tensor Flow [34]. Objective C tend to have fewer duplicates followed by the C++ programming language. Data and trends are provided fig 5.4 and 5.5 below.

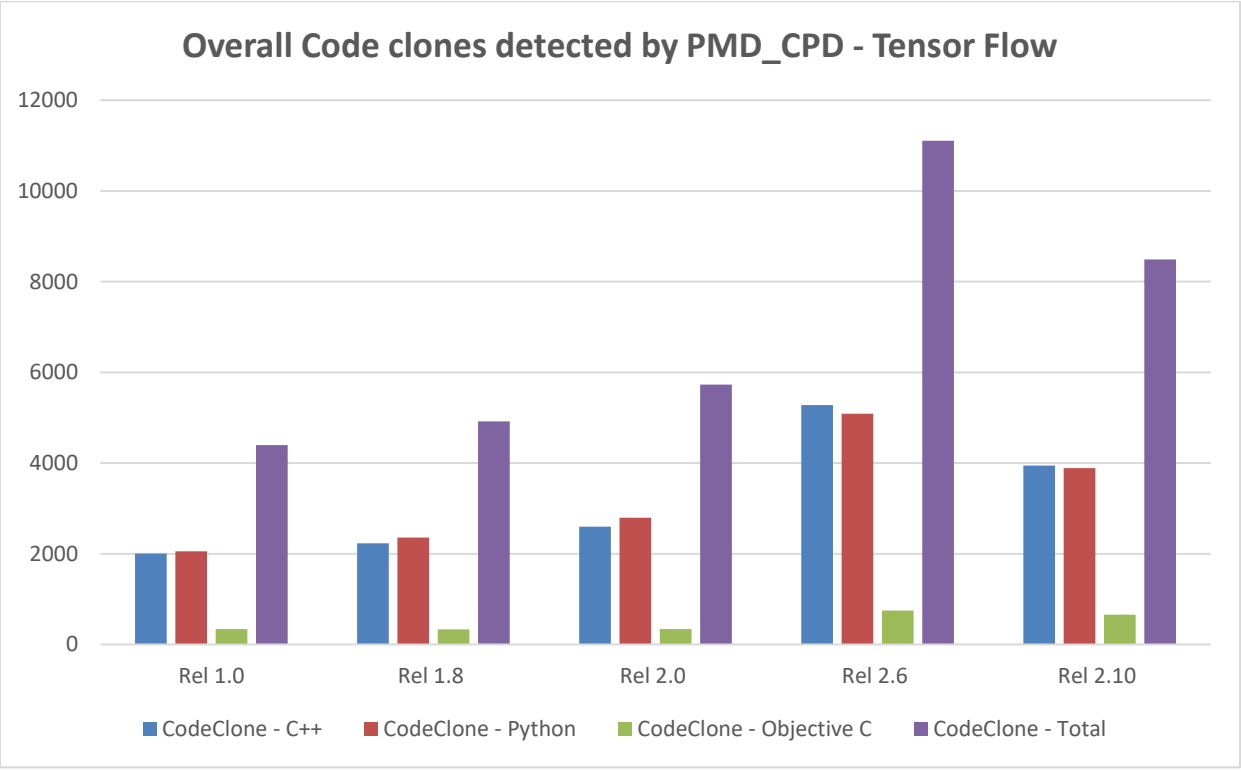


Fig: 4.15 Duplicate code detected by the PMD-CPD in Tensor-Flow

Table: 4.2 Number of code clones detected by the PMD-CPD in Tensor-Flow

Tensor Flow Version	Code Clone - C++	Code Clone - Python	Code Clone - Objective C	Code Clone - Total	Code clone percentage
Rel 1.0	2003	2052	341	4396	0.09228945
Rel 1.8	2227	2359	331	4917	0.091406104
Rel 2.0	2596	2791	342	5729	0.100868572
Rel 2.6	5278	5086	745	11109	0.117003119
Rel 2.10	3943	3889	655	8487	0.111479757

4.2 Key Findings

In this particular section, we are going to discuss about our key findings obtained by analyzing the result.

Percentages of violations in Research Codebases are significantly higher.

There is huge percentage of violation has been detected when we talk in terms of "QGIS" a code base built by researcher. The percentage of violation is calculated by dividing the number of class/methods exceeding the threshold value by the total number of the class, for this purpose, we have extracted the list of method and classes by filtering out the report-based on length of code.

It has been observed that in research codebase, there are large number of methods exceed the threshold value when compare to its overall code base size and it is growing in very latest release. On the other hand, the growth of large methods in Professional codebase is much lower and stagnant across various releases. You can reference that in figure 4.16.

Apart from the large method, there is higher number of large class detected in the research-based codebase than in professional. These data trends shown in fig 4.16 validates our quantification measurement correctness. We have observed the similar while calculating the code smell value and density.

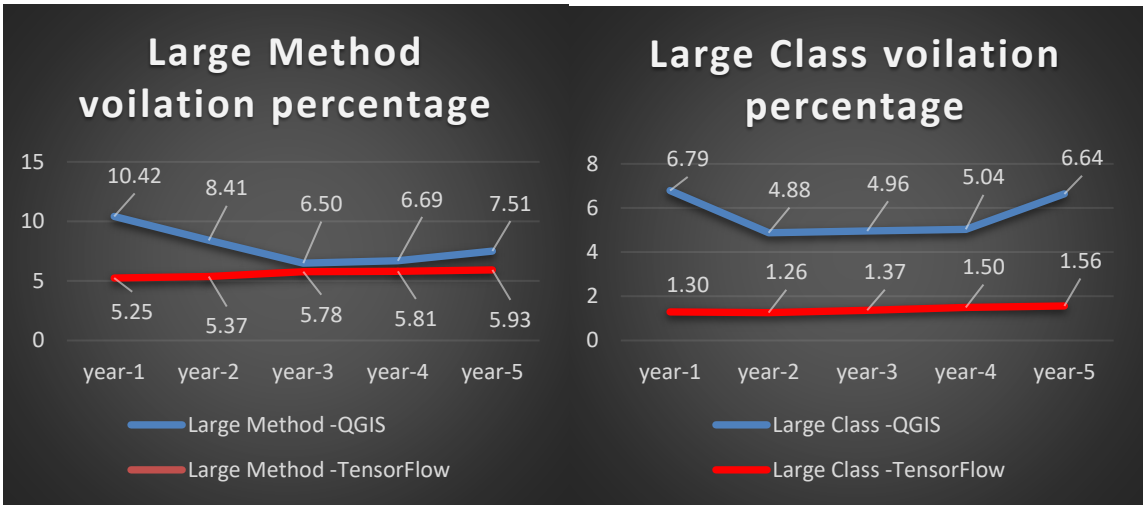


Fig: 4.16 Large Method/class Violation Percentage

Professional codebases' units tend to be significantly smaller in size.

Research codebases quality is significantly lower than comparable professionally developed codebases and the declination in quality is more severe. We have observed that in research-based codebase, there are percentage violation of large class and method are more than professional codebases. Due to this issue, there is an urgent need of appropriate refactoring process. Besides that, there is decline in the number of Small class and method add the maintainability issue in the long run. It has been observed that research-based code base has the class which possess very low comments ratio. On other hand, professional codebase posses smell way lower than that of QGIS.

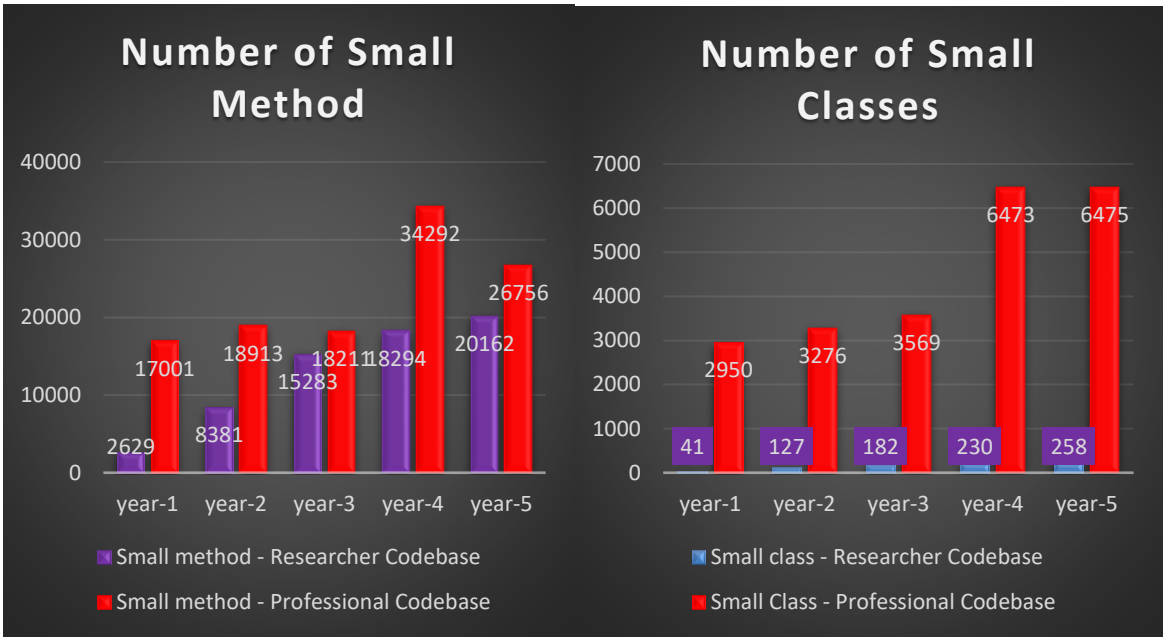


Fig: 4.17 Number of Small method and Small classes

Professional codebases suffer from too much documentations.

Professionally developed codebases tents to be significantly more documented than research codebases. This may be due to organizational requirements, or because the thresholds to quantify documentation is inadequate. As you can see in Fig: 4.18, there are large number of the excessively commented classes in professional codebase than in the research codebase. It is very serious that something is very wrong more than half of the code is commented. The classes that suffers from the excessive comments are growing over period of time while the classes in research codebase are comparatively very low. It is been observed that the research code base has the classes with are very low comment ratio. The may be due to lack of proper coding standard instruction document or code review.

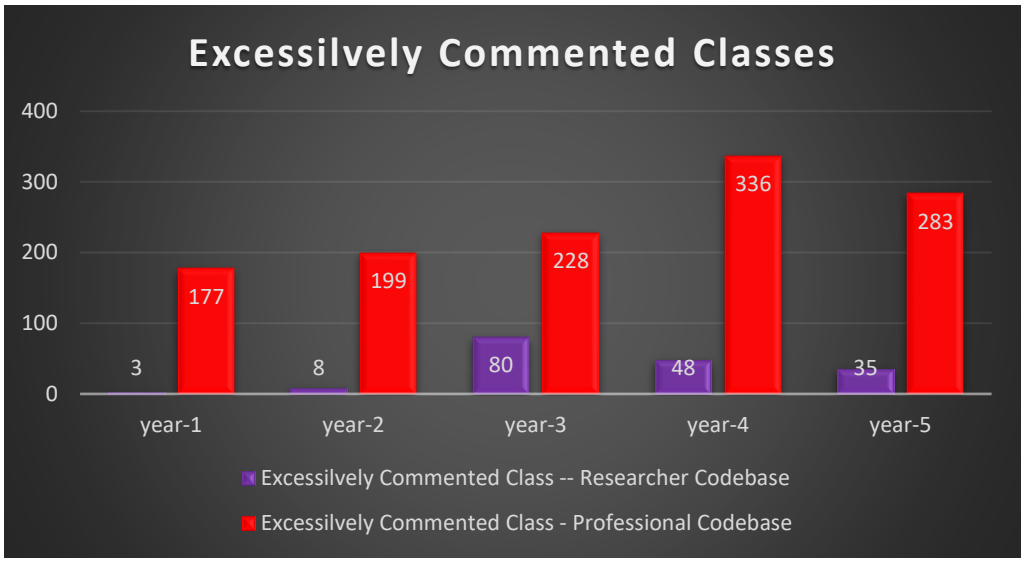


Fig: 4.18 Number of excessively commented classes

We observe a significant increase in certain smells in professional codebase overtime.

Some smells are more evident in professionally developed codebases, such as Small Class and small method code smell. This allows the codebase to grow overtime with fewer violations.it gives an indication that the classes or method which possess few lines of code could help to reduce refactoring efforts in a long run. Increase in the small classes and method is possibly due to introduction of new features that are yet to be implemented.

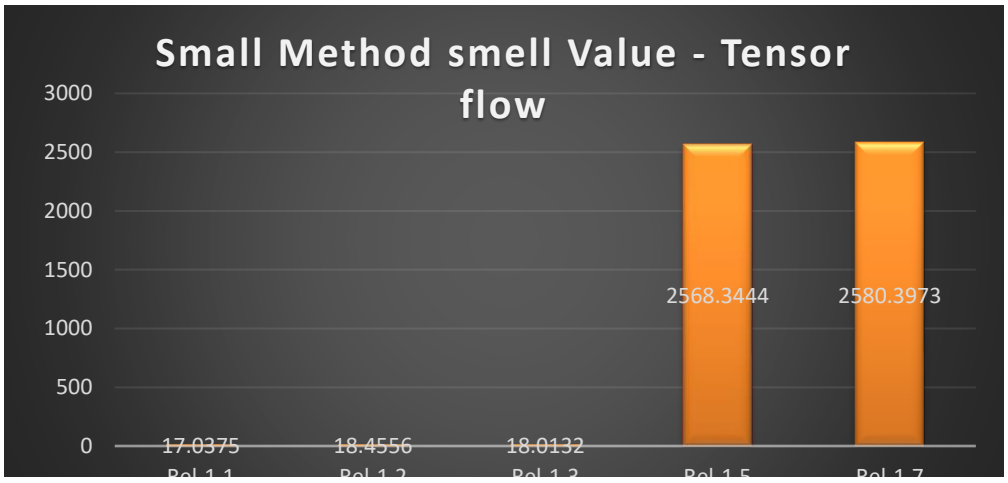


Fig: 4.19 Small Method Value in Tensor Flow

As per data extracted from the Research and professional codebases, we have observed that there is heavy increase in small class and small methods in the professional codebases.

C++ has the highest code clone measures in both professional and research codebases. Objective C had the lowest code clone values in both codebases.

C++ has highest number of code clones in both the software. C++ constitutes almost 70% of the entire Research and Professional codebases and that the reason, we have large amount of having large amount of duplicates codes. It is a fact that few languages like python requires less code to perform an operation while on the other hand, C++ require more lines to complete the same operation.

This could be other reason of having more C++ code clones. Similarly, Object C contribute very less of entire codebase so this could be one of the reasons of having fewer duplicate codes.

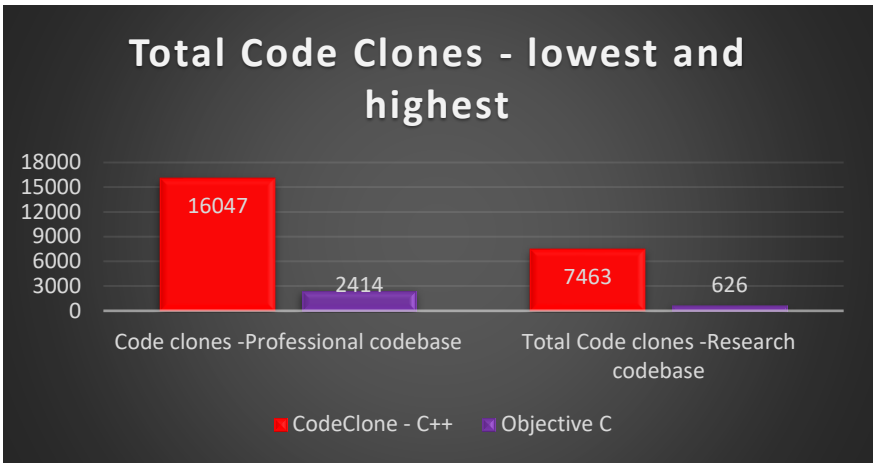


Fig: 4.20 Number of highest and lowest clones

Code clones smell was higher in professional codebases and increasing more rapidly compared to research codebases.

This is one of unexpected phenomenon which we have observed in professional codebase that it suffers a greater number of duplicate codes. In a long run, the code clones are definitely going to increase the overall maintenance cost of a software and it also impact the software quality. it is defect prone as making the inconsistent changes to code may break the functionality or cause unpredictable error. It is better to use a generic method so that it can be reuse the peer programmer to implement the functionality. however, the reason behind the such massive increase in the code clones are not clear. As software grows, it would be hard to maintain since making a change may require an additional effort to change at multiple places.

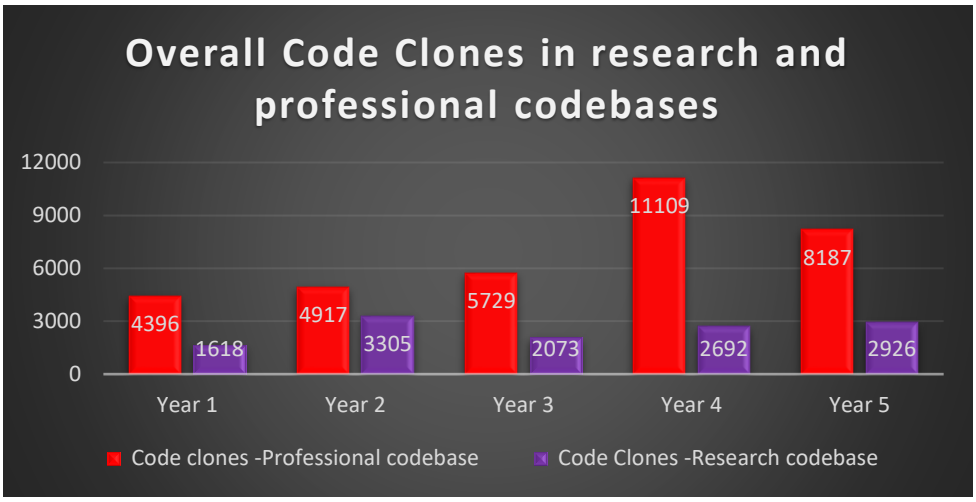


Fig: 4.21 Overall Clones detected in research and professional codebases

4.3 Validating the Code smell Quantification

To Validate our quantification measure, we have selected small set of programs namely Pricewatcher-Andriod, Pricewatcher-Java and Pricewatcher-Lib so that we can evaluate the code smell detected by our smell measurement formula is correct. In order to so, we have used Static analysis tool understand to get the basic information like total number of classes, method along with their total length so that we can use this information in our smell measurement formula to calculate the code smell value and density. We have observed there are 40, 31 and 24 classes in Pricewatcher-Andriod, Pricewatcher-Java and Pricewatcher-Lib respectively. We then use these values for calculating code smell density and value.

The dataset which we have taken for our analysis is very small, hence I have verified the class/method code length manually for all three datasets namely Pricewatcher-Andriod, Pricewatcher-Java and Pricewatcher-Lib against the data obtain from the static analysis tool "Understand" [32]. We have observed that the tool Understand is able to detect Overall LOC present in the Classes and methods precisely. As you can see in the fig:22, there is snapshot

taken from program editor which shows that the class “Notification” possess 6 lines of code and it is verified from the data taken by the Tool as seen in Fig: 4.23.

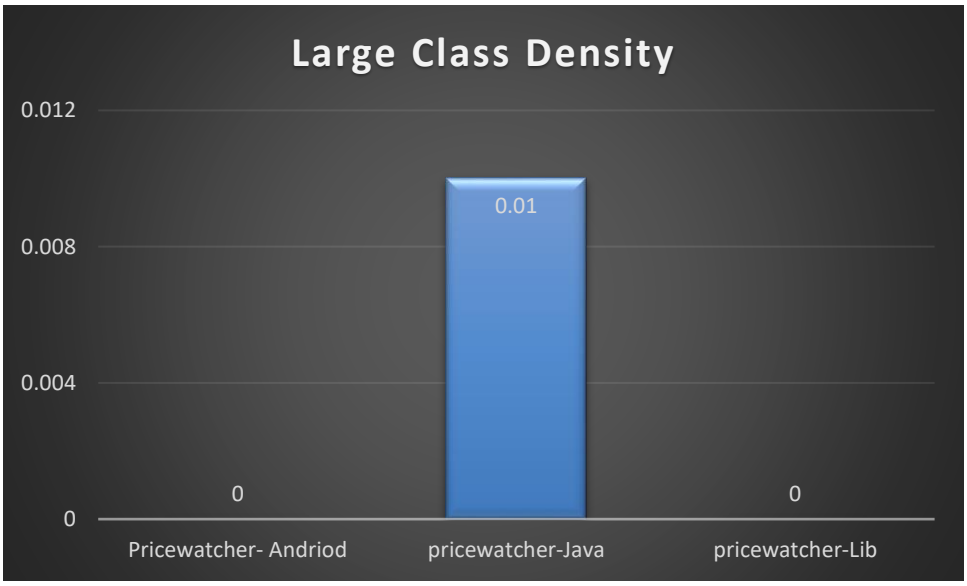
```
192     public static class NotificationID {
193         private final static AtomicInteger c = new AtomicInteger(0);
194         public static int getID() {
195             return c.incrementAndGet();
196         }
197     }
```

Fig: 4.22 Snapshot of program “BaseActivity.java” from program editor

A	B	C
Kind	Name	Total Lines
Public Static	edu.utep.cs.mypricewatcher.BaseActivity.NotificationID	6

Fig: 4.23 Snapshot from static analysis Tool “Understand” displaying Total LOC

We have calculated the code smell density and value by using our code smell measurement formula as mentioned in the Chapter 3. As you can see in fig:4.24, there are no large classes present in the Pricewatcher-Andriod and Pricewatcher-Lib while on the other hand we have observed code smell value and density in Pricewatcher-java. It indicates the presence of large class in pricewatcher-java and it is verified from the static analysis tool report.



E	F	G
Kind	Name	Total-Line
Public Class	pricewatcher.ui.Main	790

Fig: 4.24 Large Class density and value for Price watcher dataset

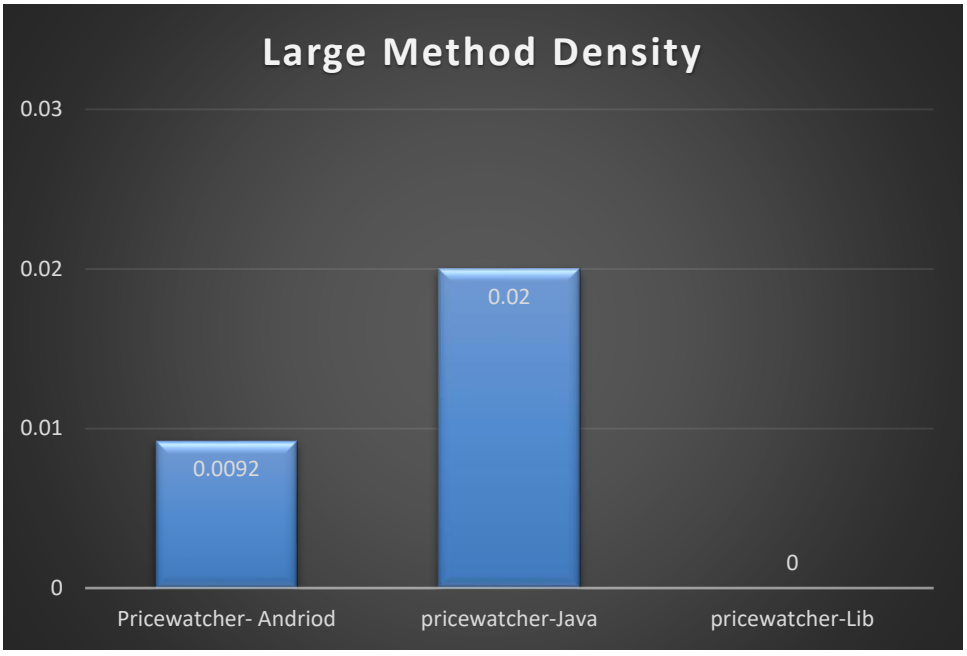
We have verified the code smell values by looking into the specific program "pricewather.ui. main" to verify that code smell detected by our quantification measure are correct. In fig 4.25, there is a snapshot from the specific program which clearly indicates that the class "main" qualifies for a large class by exceeding the threshold value

```
064 /**
065 * A dialog for tracking the prices of items.
066 *
067 * @author Yoonsik Cheon
068 */
069 @SuppressWarnings("serial")
070 public class Main extends JFrame {
071
072     /** Default dimension of the dialog. */
073     private final static Dimension DEFAULT_SIZE = new Dimension(400,
074
075     protected final static String ALL_ITEMS = "All items";
076
077   private final static StoreMenuItem[] stores = {
078       new StoreMenuItem(ALL_ITEMS, null),
079       new StoreMenuItem("Amazon", "amazon.com"),
```

```
847             JOptionPane.showMessageDialog(this, label, "Abour",
848                 JOptionPane.PLAIN_MESSAGE);
849         });
850
851     protected static final AppAction NO_ACTION
852       = new AppAction(null, null, null, 0, null);
853
854 }
855 >> EOF <<
```

Fig: 4.25 Snapshot from the Editor of a Class "Main.java"

Below is the large method density calculated for all the three program packages. It has been observed that pricewatcher-java shown large amount of code smell value and density. It has been verified by looking into the number of large methods present in all three packages of the pricewatcher. Similar to the large class and method smells, I have verified data for rest of the code smell present in patchwatcher.



	No of Large Method	
Pricewatcher- Andriod	pricewatcher-Java	pricewatcher-Lib
1	4	0

Fig: 4.26 Number of large methods in Price watcher-Java

Apart from the price watcher, I have extracted the list of classes and methods from QGIS and Tensor-Flow that are violating the threshold value and qualifying for the code smells for over certain period of time. Once the number of classes are retrieved, we have divided it the total number of classes present in that particular version to get the overall violation percentage. From the fig: 4.27, we have observed a similarity between the graph stating the growth of large method/classes in an opensource software and graph stating the code smell value and density. Both of the graphs are showing the exact same trends, although the number scales of measurement are different, but it shows that the applied quantification measure are measuring the code smell value and density very precisely.

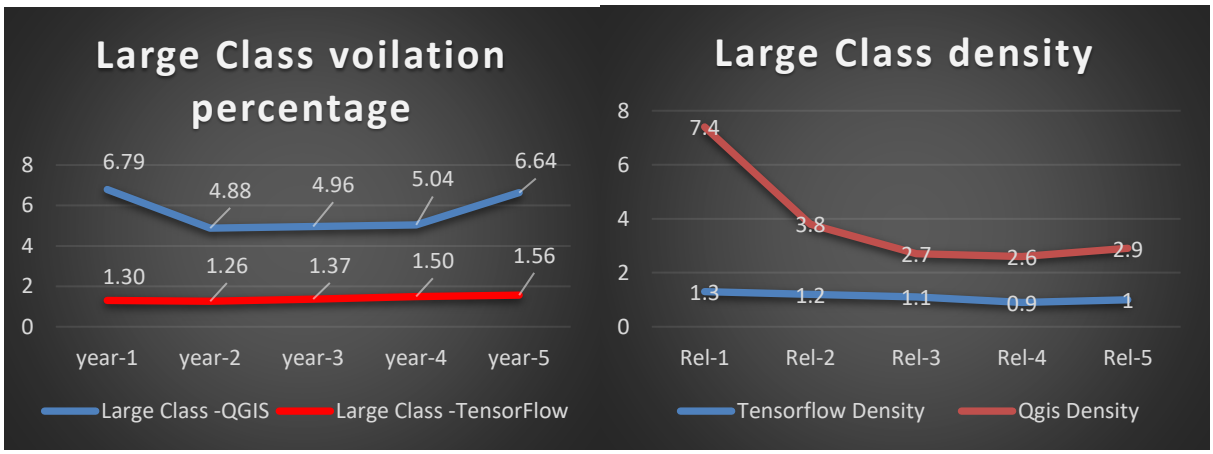


Fig: 4.27 Number of large class violation and density comparison

There is various other quantification measure to access the code quality of a software. Technical debt is one of them. Technical debt (also known as design debt or code debt) is a concept in software development that reflects the implied cost of additional rework caused by choosing an easy solution now instead of using a better approach that would take longer. Technical debt accesses the software quality on the basis of code smells, bugs and security vulnerabilities and as an output, it represents the total time required to fix the code vulnerability. We have used the tool SonarQube[35] to calculate the technical debt and ran it on Pricewatcher-Andriod, Pricewatcher-Java and Pricewatcher-Lib respectively as shown in fig 4.28.

Fig: 4.28 Snapshot from the Tool SonarQube, showing the Required Time

The result from the SonarQube [35] is stated below in the fig 4.29, it has been observed from the result that price watcher- java possess more smells and require more fixation time than price watcher-Android and price watcher-lib. There are more code vulnerability observed in the price

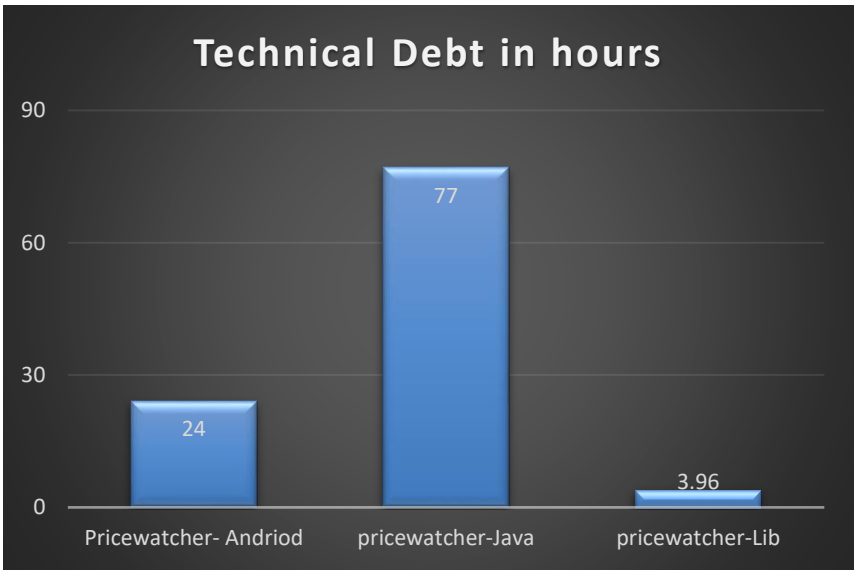


Fig: 4.29 Technical Debt in PriceWatcher Dataset

watcher- java when we analyzed by using the SonarQube [35] and these results seems aligned by the results taken from our Quantification measures. By applying our quantification, we have observed that the code smell value for pricewatcher- java is more than pricewatcher-lib and pricewatcher-Andriod in most of the cases. Technical debt results taken from the SonarQube [35] clear verifies our novel smell measurement formula's.

CHAPTER 5: CONCLUSION AND FUTURE WORK

The software market is growing with a large pace and have become very competitive which in turn have increased the customer expectation from the software industry. Software industries investing a huge amount of money in increasing the quality of the product.

Historical information with respect to source code likewise reflects structural choices by recording the evolution of the design. Furthermore, it is significant in the appraisal of maintainability. A few dependable methodologies have been created keeping in mind the end goal to recognize changes and refactoring that has been connected amid the historical backdrop of Software. In our thesis, we will quantify the code smell and present the results which we have obtained by applying our measure on an opensource software. we have studied the evolution of various code smell related to coding and design issues.

The findings which are being talked about in our thesis, at a begin with level can be considered as project related, as in they describe parts of the code and design quality for the specific opensource software that have been analyzed. Be that as it may, they additionally give starting proof with respect to the refactoring Practices that have been taken after amid the historical backdrop of the inspected software.

5.1 Future Work

In our work, we are measuring the bad smell based on the code length but there are many other parameters-like classes/methods coupling, cohesion, and cyclomatic complexity. At a later time to strengthen our code smell quantification, we will incorporate all the mentioned parameter in our formula.

At present, we are using QGIS and Tensor-Flow as our case study. We will broaden our scope of study by considering other GUI standalone and Web-based application to compare and calculate a greater number of metrics to detect the more bad smell. We are trying to build the quantification measure for the smells like feature envy, large parameter list, shotgun surgery, message chains, and dead code. By adding more software as a use case would help us to validate our new measures.

Along with the bad smell quantification, the refactoring method would be applied on a bad smell to know the severity of it.

5.2 Summary

This Paper has presented the measures to quantify the code smell by using opensource software as a use case. Code smell is often a sign of code degradation and size of the program often make peer programmers understand the code. We have presented the measurement for 7 code smells namely large class, large method, small class, small method, large comment ratio, small comment ratio, and code clones. The consolidated table shows the code smell value and density of each software QGIS and Tensor-Flow. Calculated metrics value helps the developers to apply the proper refactoring method to neutralize the severity of bad smell. The work is done in the thesis also draw a comparison between professionally developed software and software developed by the researchers on the basis of bad smell value and density. Using two different opensource systems helped us to validate our quantification measure. The Study suggests that the quality of the software developed by the researcher declines over time. Experiment data collected from the study shows major to worry about the software maintainability.

Static analysis tool (Understand [31] and PMD-CPD [32]) were used to collect the data from various opensource software version. 15 reports were generated from Tool Understand and 3 reports perversion were generated from PMD-CPD [32] which is further used in our measurement. Overall seven bad smell were identified from 15 version of QGIS and Tensor-flow. Comparative analysis was performed on the result dataset obtain from various version of opensource software and it is been observed that professionally developed software possess less code smell than software developed by the researchers. Bad smell in QGIS is growing very rapidly over a period of time while on the other hand code smell value and density in Tensor-Flow is very less and declining. Code smell could be one of a many factor to reduce the overall quality of the program and by addressing the issue in the project development process could help us to maintain the software in a long run.

Maintenance of legacy software is one of the major concerns of most IT companies today. By including the quantification measure in their respective code review process would help them to overcome the design and code related issue. In a long run, the companies could save a lot of funds if they would have taken the decision in the early stages.

5.3 Key Contributions of this Research Work

Opensource Software market is growing with large pace and have become very competitive which in turn have increased the customer expectation from the software industry. It is not mandatory for a person to have formal software training to contribute to an opensource project. Some people usually join an opensource project to improve their coding skills and that could help them in their respective career and that's the reason why quality is always crucial factor on which

many researchers is working on. Software industries investing huge amount of money in the increasing the quality of the product so that it can be sustain for long period of time.

The findings which are being talked about in our thesis, at a begin with level can be considered as project related, as in they describe parts of the code and design quality for the specific opensource software that have been analyzed. Be that as it may, they additionally give starting proof with respect to the refactoring Practices that have been taken after amid the historical backdrop of the inspected software. Most of literature present currently talks about the code smell detection, they have used binary system which counts the number of classes or methods violating the code smell rules and reporting it accordingly but none of them discuss about the severity of code smell in a code. For an example there are metrics suggested by various literature: [1] weighted method count which is nothing, but the static complexity of all method presents in the class is greater than or equal to 47 or tight class cohesion > 1/3 or total counts of foreign attribute which are accessing the class in some way is greater than 5. There are many more literature which focused on determining the code smell based on the line count, method count, coupling ratio and cyclomatic complexity.

Our research is based on quantification measure which not only talks about smell detection but also talks about the severity of code smells in an opensource software. We have used an exponential function which check the severity of code smell by measuring the distance of a class or a method from the threshold value. As the distance from the threshold increases, the code smell severity increased along with that. Our quantification measurement not only detects the code smell by using the threshold value but also penalize the class or a method based on their distance from the threshold. By using this approach, we understand the evolution of code quality characteristics in a researcher based opensource software and professional developed software. we understand

from our results that if the code smell is detected in very earlier releases, then it takes next two or three releases to mitigate the smell through proper refactoring process.

By shedding light on these principles and results, managers and moderators of software companies can device better strategies for improving software quality and sustainability. We can use this approach in an organization software development and maintenance process to improve the overall code quality of a software. We will be strengthened the smell measurement formula by involving other critical factors that affect the quality characteristics. We will be building quantification measure for other code smell.

To Summarize, this dissertation talks about the quality characteristics and their evolution related to an opensource software. The principal contributions of this thesis are:

- We introduce a model that supports performance by understanding the severity of code smell in research based opensource software. This model becomes the foundation of this thesis research, which is used as a tool for supporting quality analysis.
- Our quantification measure would help in accessing the software quality and detecting the code smell in very early releases of software. This would help programmers to sustain a software for a longer period (As a literature [9] says that code smell grows as a project mature. If a project is poorly coded and badly designed, then it is definitely going to impact in a long run.
- We have analyzed the evolution of quality characteristics in an opensource research based software and professional codebase. It has been observed that research-based codebase tends to have more code smells than professional codebases.
- We have highlighted a novel technique to quantify the seven types of code smell.

- Our highlighted technique could help the programmers to reduce and rectify the number of issues in a software.
- From our experimentation, it turns out that Large method and large class code smell are very prominent in Research based codebases and it is growing over time while professional codebase has large number of small methods and classes.
- From our experimentation, it turns out that professional code base like to suffer with excessive documentation while on the other hand, the research code base has very poorly documented code. This may be due to organizational requirements, or because the thresholds to quantify documentation is inadequate.
- Our quantification measure can be used in the software development process to improve the code quality.

REFERENCES

[1] S. Olbrich, D. S. Cruzes, V. Basili and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," 2009 3rd International Symposium on Empirical Software Engineering and Measurement, Lake Buena Vista, FL, 2009, pp. 390-400.

[2] Mika V. Mäntylä. An experiment on subjective evolvability evaluation of object-oriented software: Explaining factors and interrater agreement. In Proceedings of 2005 International Symposium on Empirical Software Engineering, pages 287–296, Noosa Heads, Queensland, Australia, November 2005. IEEE Computer Society. doi:10.1109/ISESE.2005.1541837.

[3] S. M. Olbrich, D. S. Cruzes and D. I. K. Sjøberg, "Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems," 2010 IEEE International Conference on Software Maintenance, Timisoara, 2010, pp. 1-10.

[4] K. Dhambri, H. Sahraoui & P. Poulin, (2008) “Visual Detection of Design Anomalies”, Proceeding of 12th European Conference in Software Maintenance and Reeng, pp279-283.

[5] Chidamber, S. R., & Kemerer, C. F. (1994). A Metrics Suite for Object Oriented Design. IEEE TSE, 20 (6), 476-493.

[6] M. V. Mantyla, J. Vanhanen and C. Lassenius, "Bad smells - humans as code critics," 20th IEEE International Conference on Software Maintenance, 2004. Proceedings., 2004, pp. 399-408.

[7] A.Yamashita and L. Moonen, "Do developers care about code smells? An exploratory survey," 2013 20th Working Conference on Reverse Engineering (WCRE), Koblenz, 2013,

pp. 242-251.doi: 10.1109/WCRE.2013.6671299

[8] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto and A. D. Lucia, "Do They Really Smell Bad? A Study on Developers' Perception of Bad Code Smells," 2014 IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, 2014, pp. 101-110.

[9] Chatzigeorgiou and A. Manakos, "Investigating the Evolution of Bad Smells in Object-Oriented Code," 2010 Seventh International Conference on the Quality of Information and Communications Technology, Porto, 2010, pp. 106-115.

[10] F. Khomh, M. Di Penta and Y. G. Gueheneuc, "An Exploratory Study of the Impact of Code Smells on Software Change-proneness," 2009 16th Working Conference on Reverse Engineering, Lille, 2009.

[11] Patricia S. Abril and Robert Plant. 2007. The patent holder’s dilemma: Buy, sell, or troll? Commun.ACM 50, 1 (Jan. 2007), 36–44. DOI:http://dx.doi.org/10.1145/1188913.1188915

[12] T. Vale and I. S. Souza, “Influencing Factors on Code Smells and Software Maintainability: A Cross-Case Study,” in 2nd Workshop on Software Visualization, Evolution and Maintenance, 2014

[13] Katzmarski, B. and Koschke, R. (2012). Program complexity metrics and programmer opinions. In Program Comprehension (ICPC), 2012 IEEE 20th International Conference on, pages 17–26.

[14] F. A. Fontana, E. Mariani, A. Mornioli, R. Sormani and A. Tonello, "An Experience Report on Using Code Smells Detection Tools," 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, Berlin, 2011, pp. 450-457.

[15] Liu, Xinghua & Zhang, Cheng. (2017). The detection of code smell on software development: a mapping study. 10.2991/icmmct-17.2017.120.

[16] M. J. Munro. Product metrics for automatic identification of" bad smell" design problems in java source-code. In 11th IEEE International Software Metrics Symposium (METRICS'05). IEEE, 2005.

[17] Rasool, G., Arshad, Z., Nov. 2015. A review of code smell mining techniques Journal of Software: Evolution and Process 27 (11), 867– 895.

[18] Fowler, M., 1999. Refactoring: Improving the Design of Existing Programs,1st Edition. Addison-Wesley Professional.

[19] Brown W. H., Malveau, R. C., McCormick, H. W. S., Mowbray, T. J., 1998. AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis, 1st Edition. John Wiley & Sons, Inc.

[20] A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok, "Refactoring test code," in Extreme Programming Perspectives, M. Marchesi, Ed., 2001, pp. 92-95, Addison-Wesley, 2002.

[21] G. Suryanarayan, G. Samarthyam, T. Sharma "Refactoring of Software design smell: Managing Technical Debt". Morgan Kaufmann, 2014.

[22] M.Lippert, S. Roock, "Refactoring in Large software Project: Performing the Complex restructuring successfully" John Willey and sons, 2006.

[23] Van Deursen, Arie & Moonen, Leon & Bergh, Alex & Kok, Gerard. (2001). Refactoring Test Code.

[24] Kaur S, Maini R (2016) Analysis of various software metrics used to detect bad smells. Int J Eng Sci (IJES) 5(6):14–20

[25] Anshu Rani and Harpreet Kaur "Detection of bad smells in source code according to their object-oriented metrics", International Journal for Technological Research in Engineering

Volume 1, Issue 10, 2014.

[26] Sandeep Kaur and Harpreet Kaur "Identification and Refactoring of Bad Smells to Improve Code Quality", International Journal of Scientific Engineering and Research, Volume 3, Issue 8, August 2015.

[27] Ph.D thesis by Kwankamol Nongpong, University of Wisconsin-Milwaukee "Integrating Code Smells Detection with Refactoring Tool Support", August 2012.

[28] P. Danphitsanuphan and T. Suwantada, "Code smell detecting tool and code smell-structure bug relationship," in Proc. 2012 Spring Congr. Eng. and Technology (S-CET). Xi'an, China: IEEE, May 2012, pp. 1–5.

[29] Williams, Laurie, Dright Ho, and Sarah Heckman. (2005). SoftwareMetrics in Eclipse [Online]. Avaliable: http://agile.csc.ncsu.edu/SEMaterials/tutorials/metrics/

[30] Swapnil Chauhan, Badreddin, Omar, Wahab Hamou-Lhadj. "Susereum: Towards a Reward Structure for Sustainable Scientific Research Software". In proceedings of Software Engineering For Science (SE4Science), collocated with ACM/IEEE International Conference on Software Engineering, 2019.

[31] Understand – Source Code Analysis and metrics, https://scitools.com/trial-t hanks/understand/

[32] PMD's and tool, 2009 PMD's CPD tool, Retrieved October 10th, 2018 from <http://pmd.sourceforge.net/cpd.html>.

[33] "QGIS Development Team (2018). QGIS Geographic Information System. Open Source Geospatial Foundation Project. http://qgis.osgeo.org

[34] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat,

Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[35] SonarQube: Continuous Inspection

https://sonarcloud.io/about/sq

CURRICULUM VITA

Educational Background

My name is Swapnil Singh Chauhan. Currently, I am pursuing the master's in computer science at the University of Texas at Elpaso. I have completed my bachelor's degree in computer science in 2012 from Rajiv Gandhi Technical University, India.

Professional Experience

I possess 5 Years of extensive experience in software research and development. I am Expert in advanced development methodologies, tools, and processes contributing to the design and rollout of cutting-edge software applications. Previously, I used to work at Infosys private limited in India for two and a half years. My role as a software developer is to analyze code and proposing well engineered cost-effective solutions. I was also responsible for mentoring freshers joining the company about Insurance & mortgage domain knowledge, product functionality, Business process models, the creation of technical documentation and ways to create workflows in the system and write optimized code. Before Infosys, I had worked in another multinational company; Altisource business solution for approximately two and a half years as Junior software engineer. My role as developer is to build new modules and functionality for loan/policy servicing product.

Academic Experience

I had worked as a research assistant at university of Texas at Elpaso. Currently, I am working on quantifying the code smell and study the smell evolution in an opensource software.

Contact Information: sschauhan@miners.utep.edu

This thesis/dissertation was typed by Swapnil S Chauhan