Empirical Study of Long Parameter List Code Smell and Refactoring Tool Comparison

Saira Moin u din[1], Fatima Iqbal[2], Hafiz Ali Hamza[3] and Sajida Fayyaz[4]
[1,3,4]Department of Computer Science, University of Lahore, Pakistan (Sargodha Campus)
[2] University of South Asia Lahore, Pakistan
[1]saira.moin@cs.uol.edu.pk, [2]fatima.iqbal313@gmail.com, [3]alihamzagondal12@yahoo.com, [4]sajida.fayyaz@cs.uol.edu.pk

Abstract– **The main focus of software engineering industry is performance, security and reliability which are difficult to manage in software at the same time. The main hurdle to achieve this is code smells that hinders the performance of software. Martin Fowler defined 22 bad code smells and their treatment is termed as refactoring. Refactoring improves the overall structure of the software and results an overall increase in quality of a software. There are different tools in market for code smell detection and refactoring but none of the tools can treat all code smells. We have presented a java based prototype BSDR for bad smell detection and refactoring based on the principal of human mental theory. We have compared the results of BSDR with two market oriented tools, Checkstyle and PMD (source code analyzer) against a code smell named Long Parameter List. The results show that PMD and Checkstyle show almost same results but BSDR shows little bit better results as compare to both which can be better in future.**

Keywords– **Code Smells, Refactoring, BSDR (Bad Smell Detection and Refactoring) Long Parameter List, Checkstyle, and PMD**

I. INTRODUCTION

Smell detection and refactoring usually lies under software quality that mainly targets software maintenance and extensibility along with the other quality attributes. Code smell detection and refactoring goes side by side. Where code smells, termed as bad code are design flaws in source code. These smells indicate that something somewhere in code has gone off beam. In [3] smell taxonomy was presented. Fowler et al., [1] recognized 22 bad smells going from a simpler bad smells like" code duplication" and "Long parameter list" to more complex smells like "God Class" and "Feature Envy". He has also defined 22 bad smells into seven classes centered on the basis of resemblance. Refactoring is a technique which can be manual and automatic for the treatment of code smells in a source code of program.

Refactoring has become an important practice in the software development and maintenance. According to Martin Fowler and co-authors the process of refactoring provides an improvement in the internal structure of a program without changing its external behavior. Refactoring a code makes it easier and understandable for a programmer and enhances its quality and design [3]. Different Refactoring tools and plugins are available in market but none of them can provide complete access to all bad smells purposed by different researchers. According to Tom Mens and Tom Tour [4] refactoring is a step by step procedure, described in six activities:

- Recognizing where refactoring is needed.
- Defining the type of refactoring(s) need to apply to the recognized places.
- Assuring that the applied refactoring preserves behavior.
- Apply selected method of refactoring.
- Evaluating refactoring effects on software process and quality.
- Measuring consistency related to code and other software artifacts.

Different refactoring techniques are introduced by different researchers such as:

- More abstraction
- Breaking codes into logical pieces
- Improving name and location of code

There are different refactoring tools in the market like jDeodorant [26], CheckStyle [6], PMD [7], SolidSDD [8], InCode [9], JRefactory [10] and many others but all of these provide refactoring for few code smells and use different approaches. Checkstyle [6] is an eclipse plug. It checks code layout issues and many other checks have also been provided for different purposes. Checkstyle offers checks that find class design problems, duplicate code, or bug patterns.

IntelliJ IDEA [11] a java Integrated Development Environment, which besides many other features also detects duplicates and provides refactoring support. PMD [7] is static analysis tool that find dead code, duplicate code, overcomplicated expressions and more. It is integrated with "JDeveloper", "Eclipse", "JEdit", "JBuilder", "BlueJ", "NetBeans", "IntelliJIDEA", TextPad, Maven, Ant, Gel, JCreator, andEmacs.

In this paper, we have presented a prototype, designed in java for code smell detection and refactoring using a different approach regardless of the above mentioned tools. We have presented the results and as well a comparison of the results. Where our prototype is showing little bit better results than the other selected tool.

II. LITERATURE REVIEW

Khomh et al. in [25] presents that class with code smells is morelikely to change then the classes without code smells. They further show the correlation between particular type of code smell and change proneness. They use DECORE (Defect dEtection for CORrection) [24] approach to detect code smells and apply changed mathematical techniques to achieve the outcomes. Two open source systems from different domains were chosen for experiment one was "Azureus" and the other was Eclipse used by both open source community and industry. From 13 releases of Eclipse and 9 releases of "Azureus" results showed classes with smells are more change prone then the classes without smells. They further show that particular kind of code smells lead to change-proneness. However, this study was limited to only few systems and does not consider the type and amount of change. It might give different results when used for more systems.

A similar study was presented in [5]. The basic purpose of the study was to cheek the lifetime of code smells and whether they are removed from refactoring. They experimented on two open source projects and JDeodrant was used as smell detection tools. Focus was on only three bad smells namely "Long Method", "Feature Envy" and "State Checking". Results showed that many of the smells usually persist in the code even till the updated versions of the system. Some of the smells that are removed were not a result of targeted refactoring rather they are removed as maintenance effort.

In [12] three case studies presented to show that why smell suppression is not frequently applied by the developers and why it's not a subject of significant research? In first study the subject system was five open source java based systems, the second study was on C# web based application and the third one was a theoretic record of smell-related refactoring. The study showed that one of the main reasons, why smell eradication is not applied, is associative nature of refactoring. Usually when a refactoring is applied it required another refactoring which in turn require another one and so on. So developers avoid eradicating these due to massive nesting. Many conflicts, contradictions are merged when smelly a code is identified so it makes the identification of real smell exorbitant and sticky. They further showed that smells that require simple refactoring are of more interest as compared to those that required complex ones.

Rysselberghe et al., in [13] use duplication detection techniques to reconstruct the system evolution process. In [22] an approach to diagnose design problems in Object Oriented systems has been proposed. The authors said that code bad smells are structural symptoms and by finding the correlation between these structural symptoms, cause of bad design as well as its treatment can be found. They experimented on ArgoUML which is an open-source UML modeling tool. The system is java base having 220.000 lines of code. In their approach iPlasma (automated smell detection tool [14]) is used to detect code smells in the above mentioned system. Fontana et al in [23] acknowledged different smell detection tools and differences among them. Each tool detects different smells but none of the tool detects all 22 smells as described in [1]. They did experiment on different versions of object oriented open source system and presents the results. According to their results different tools may detect the same smell but using different criteria such as a tools use only number of code lines to detect large class smell while the other tool considers the size in terms of number of methods and attributes. Checkstyle [2] is an eclipse plug. It checks code layout issues, many other checks for other purposes have also been provided. Checkstyle provides checks that find class design problems, duplicate code, or bug patterns. CheckStyle is highly configurable and can be configured to support any coding standard. It implements many standards checks that find class design problems, block checks, naming conventions, bug patterns (like doubled checked locking) and many more with bad smell detection checks [15].

Code smells are design problems that may prompt refactoring. These smells can be sensed by human intuition but scalability is big issue in this case. To resolve this issue automatic detection tools are used. So far none of the tool detects all of the code smells mentioned in smell taxonomy [16]. Some of the literature regarding these tools is presented in the subsequently paragraph. Refactoring is also widely used activity to improve the quality of the software system. A comparative study is presented in [17].

The study evaluates the three techniques that are commonly used for detection of duplicate code smell. The techniques were line matching, parameterized matching and metric fingerprinting. Five cases ranging from small to medium size were chosen for scrutiny purpose. The study focused on task specific suitability of detection technique. It was concluded that line matching is only good if basic information of clones are required. The parameterized matching gives best results if it used with refactoring tools that works on method level. Metric fingerprinting well suited with method level refactoring tools. A prototype as an Eclipse plug is presented in [18]. Context sensitivity, scalability and expressiveness are main characteristics of this prototype. Whenever the user browses the underlying code, half circle of wedges (triangular shape) is appeared on smelly code. Radius of triangular shape denotes the weight of smell in current context.

When mouse is put over the founded smell, name of the particular smell is shown on a label. If further detail is required about the smell, programmer can click on the smell label. Instances of each existing smell and contributors of that smell also spotlighted by this tool. Software Maintenance is one of the SDLC activity that requires large part of software development cost (up to 80% of total budget [19] and refactoring is the activity that increase the maintainability of software thus reducing the maintenance cost. Some of the literature regarding software quality, refactoring and bad code smells and detection tools is discussed below. In [20] Steve et al presented a report of code smells analysis and claimed illusive nature of code smells which means the actual effort required to remove the smell is hidden, so the smells that seem to be eradicated easily can be difficult one to be removed. Many smells do not require single refactoring rather series of refactoring removing them from the code. He uses the [21] smell taxonomy and customized software tool to discover the figure for different refactoring techniques necessary for each of 22 code smells. Analysis shows that the smell category known as Bloaters needs a large number of refactoring to be removed from the code.

III. BSDR PROTOTYPE

Manual refactoring is time taken and error prone so we need a better tool support to refactor a code more effectively. A wide range of tool support is available in this regard but none of them provide full support of refactoring against all bad smell. We have designed a java based prototype, BSDR prototype for bad smell detection and refactoring which does not provide full support but uses a different technique that have not been used before in any of the market oriented tools. For this work we have selected open source Xtreme media player source code for as input for bad smell detection. Table I shows the details of open source Xtreme media player source code particulars.

Table I. Open source XtremeMP

Metrics	XtremeMP-0.6.6
Total LOC	7956
No of Packages	17
No of Classes	71
No of interfaces	8
No of Methods	498
Method lines of code	5142
Weighted methods per class	1341
No of static methods	70

The BSDAR is a bad smell detection and refactoring prototype that will be helpful for developers to detect bad smell and refactor those bad smells.It will also provide the refactoring option and suggestions for bad smells in order to assist the software engineers/developers to apply the appropriate refactoring techniques manually. BSDAR is implemented in Java. It provides Graphical User Interface using Java's Swing capabilities. It uses Eclipse JDT (Java Development Tools) API for bad smell detection, metrics calculation and bad smell analytics. It also uses ASTParser to construct the Abstract Syntax Tree (AST). The Integrated Development Environment (IDE) used for the development of BSDAR is NetBeans 7.1 Beta. The input of BSDAR is a Java project folder. Fig. 1 shows the working flow chart of the work done.

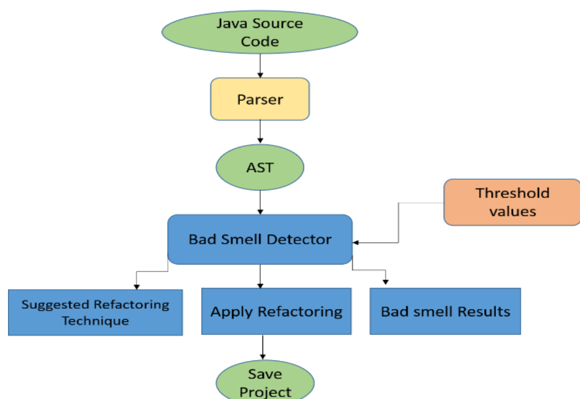

Fig. 1: Work flow of BSDR

A) Detection Technique and Algorithm

1) *Parse Java source code file and construct AST.*
2) *Visit each Method Declaration node of AST.*
3) *Count the parameters of the method.*
4) *If the number of parameters exceeds the conditions specified in our research work, consider this parameter list as "long Parameter List".*

Repeat steps from 1 to 4 for each Java source code file.

We have also added an extra parameter in the detection of "Long parameter List", risk priority level of that bad smell. So by doing this we will have a priority list which directs us for removal of high priority bad smell first and then moves towards the low levels. The threshold values were set according to human mental theory according to which a human can remember 7 items at a time. Remembering more results in lost the previous ones. Table II represents risk priority level associated with no of parameters that uses human mental theory.

Table II: Risk priority level

Impact Level	No of parameters
Low	>=3 &< 5
Medium	>=5 &<7
High	>= 7

A) Results

Results after studying the behaviour of the tools with our prototype shows that Checkstyle and PMD both use the technique of counting the number of variables passing through the definition of the method calling to detect Long parameter bad smell. It just counts the number of parameter

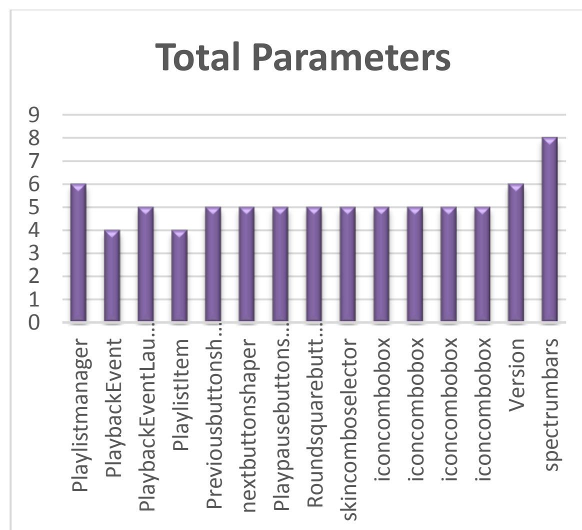

Fig. 2: PMD results based on parameters

passing in definition of method name from code and compares the value with their pre-decided threshold value which is 3. BSDAR Prototype can detect different bad smells. But we have chosen Long Parameter List only for our comparison where threshold value can be change at wish. The Fig. 2, Fig. 3 and Fig. 4 show the code smell detection results of PMD, Checkstyle and BSDR results respectively. Number of parameters is at Y axis while X axis contains information about class name. Fig. 2 shows that maximum 8 parameters and minimum 4 parameters are detected on a pre-set threshold value. Different class names are mentioned at the X axis of the graph.

Again same behaviour is shown by Checkstyle as by PMD, Fig. 3 shows that maximum 8 parameters and minimum 4 parameters are detected on a pre-set threshold value in 13 different classes. Class names are mentioned at the X axis of the graph and Y axis deals no of parameter present in a particular class. While following Fig. 4 represents the results

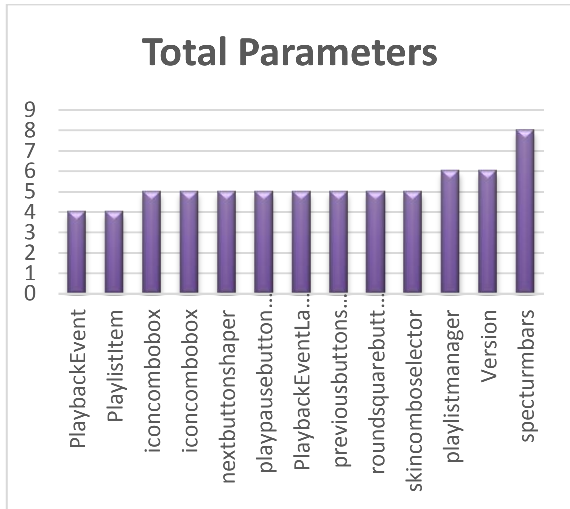

Fig. 3: Checkstyle results based on parameters

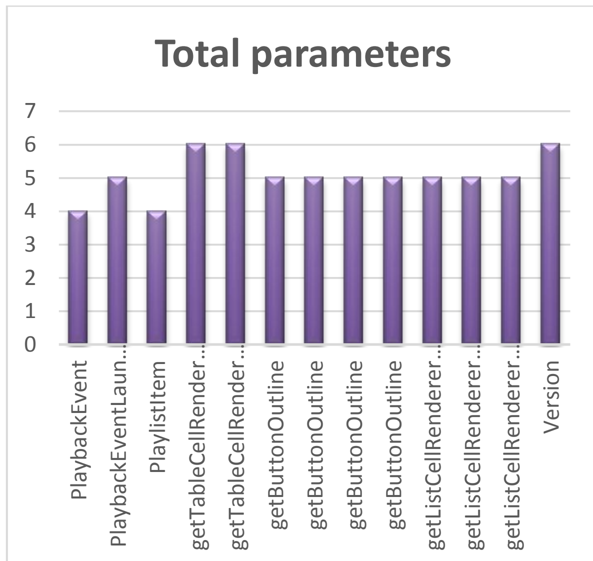

Fig. 4: BSDR results based on parameters

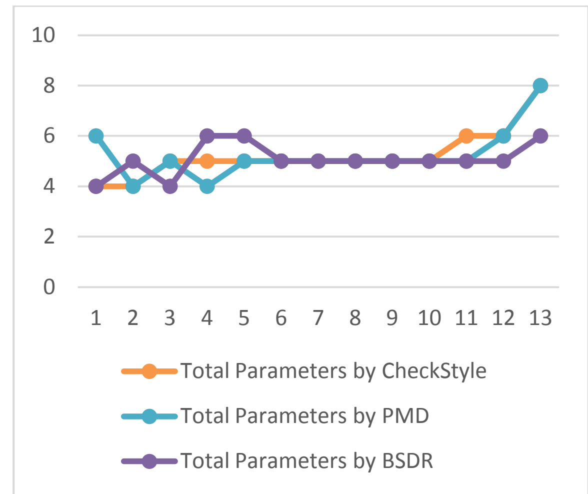

Fig. 5: Comparison of PMD, BSDR, Checkstyle

of BSDR for long parameter list bad smell, which is little bit different because a totally different approach is used in BSDR.

Compared results show that "CheckStyle" detect one extra bad smell then our prototype while "PMD" detected one less bad smell.

IV. CONCLUSION

Based on our empirical study our prototype is working efficiently for few bad code smells. We can enhance the same idea to detection of all bade smells and consider the efficiency, reliability and maintainability parameters of quality for better results.

BSDAR prototype presently supports a limited functionality. There is a possibility of many types of enhancements in BSDAR such as Aspect Oriented software development has gain a considerable attention in the recent years. BSDAR can be enhanced by making it capable of doing Aspect Oriented Refactoring for the bad smell that it detects and we can also extend this prototype to a tool with the enhancement to detect more bad code smells.

REFERENCES

[1]. M. Fowler, K. Beck, J. Brant, W. Opdyke,D. Roberts. *Refactoring Improving the Design of Existing Code*, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA 1999 ISBN:0-201-48567-2

[2]. M. Fowler, K. Beck, J. Brant, W. Opdyke,D. Roberts. *Refactoring Improving the Design of Existing Code*, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA 1999 ISBN:0-201-48567-2

[3]. M. Mantyla, J. Vanhanen, C. Lassenius. *Taxonomy and an initial empirical study of bad smells in code.* In Proceedings of the 19th International Conference on Software Maintenance (ICSM 2003). Amsterdam, the Netherlands, pp. 381-384

[4]. Coding Horror: Programming and Human Factors. *Code Smells* May 2006. http://www.codinghorror.com/blog/2006/05/code-smells.html

[5]. A. Chatzigeorgiou, A. Manakos, *Investigating the Evolution of Bad Smells in* Object-*Oriented Code,* In Proceedings of seventh International Conference on the Quality of

Information and Communications Technology (QUATIC), 2010, vol., no., pp.106-115, Oct. 2 2010.

[6]. Checkstyle Home Page, *Checkstyle 5.6* May 2012, http://checkstyle.sourceforge.net/

[7]. Eclipse:The Eclipse Foundation open source community website. *Eclipse 4.3*, June 2013, http://www.eclipse.org/downloads/packages/eclipse-standard-43/keplerr

[8]. Solid Source IT. *Duplicate Code Detection and Analysis*. June 2012. http://www.solidsourceit.com/download/SolidSDD-download.html.

[9]. Intooitus Group *inCode Helium*, April, 2012.http://www.intooitus.com/products/incode/features.

[10]. JRefactory, *Free development software download* , May 2012, http://jrefactory.sourceforge.net/

[11]. Jet BRAINS, *IntelliJ IDEA The Best Java and Polyglot IDE* , June 2013. http://www.jetbrains.com/idea/features/index.html

[12]. S. Counsell, R. M. Hierons, H. Hamza, Black, M. Durrand , *Exploring the* Eradication *of Code Smells: An Empirical and Theoretical Perspective*, Advances in Software Engineering Vol 2010, Article ID 820103, 12 pages

[13]. F. V. Rysselberghe, S. Demeyer, "*Evaluating Clone Detection Techniques From a Refactoring Prspective.* In proceedings of 19th International Conference on Automated Software Engineering (ASE) 2004, pp. 336-339.

[14]. C. Marinescu, R. Marinescu, P. Mihancea, D. Ratiu, R. Wettel. Iplasma, *An* integrated *platform for quality as-sessment of object-oriented* design*,* In Proceedings of 21st International Conference on Software Maintenance (ICSM 2005).

[15]. W. Opdyke. RefactoringObject-Oriented Frameworks. PhD Dissertation, Univ. Illinois at Urbana-Champaign, 1992.

[16]. M. Fowler, K. Beck, J. Brant, W. Opdyke,D. Roberts. *Refactoring Improving the Design of Existing Code*, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA 1999 ISBN:0-201-48567-2.

[17]. F. V. Rysselberghe, S. Demeyer, "*Evaluating Clone Detection Techniques From a Refactoring Prspective.* In proceedings of 19th International Conference on Automated Software Engineering (ASE) 2004, pp.336-339.

[18]. E. M. Hill, Scalable, Expressive, and Context-Sensitive Code Smell Display. OOPSLA Companion '08 to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications. ACM New York, NY, USA 2008 pp. 771-772. ISBN: 978-1-60558-220-7.

[19]. B. W. Boehm. Software Engineering Economics. Prentice Hall, Englewood Cliffs, NJ, 1981.

[20]. F. V. Rysselberghe, S. Demeyer, *Reconstruction of Successful Software Evolution Using Clone Detection*, Proceedings of the 6th International Workshop on Principles of Software Evolution (IWPSE) pp 126 IEEE Computer Society Washington, DC, USA 2003 ISBN:0-7695-1903-2.

[21]. C. Marinescu, R. Marinescu, P. Mihancea, D. Ratiu, R. Wettel. Iplasma, *An* integrated *platform for quality as-sessment of object-oriented* design*,* In Proceedings of 21st International Conference on Software Maintenance (ICSM 2005)

[22]. A. Trifu, R. Marinescu, *Diagnosing Design Problems in Object Oriented Systems*, Proceedings of the 12th Working Conference on Reverse Engineering (WCRE) IEEE Computer Society Washington, DC USA 2005, pp. 155-164

[23]. F. A. Fontana, E. Mariani, A. Morniroli, R. Sormani, A. Tonello, *An experience report on using code smell detection tools*, IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops. (ICSTW), 2011 pp. 450-457, ISBN 978-1-4577-0019-4.

[24]. N.Moha, Y. G. Guehenue, L.Duchien, F. L. Meur, *DECORE: A method for* specification *and detection of code and design smells.*" IEEE Transactions on Software Engineering, vol 36, pp. 20-36, Jan-Feb 2010.

[25]. F. Khomh, M. D. Penta, Y. G. Gueheneuc , *An Exploratory study of the Impact* of *Code Smells on Software Change-proneness,* In proceedings of the 16th Working Conference on Reverse Engineering (WCRE'09) 2009 , pp. 75-84, ISBN: 978-0-7695-3867-9.

[26]. T. Chaikalis, N. Tsantalis, A. Chatzigeorgiou, *JDeodorant: Identification and* Removal *of Type- Checking Bad Smells,* 12th European Conference on Software Maintenance and Reengineering (CSMR) 2008, pp. 329-331, ISBN: 978-1-4244-2157-2.