https://doi.org/ 10.33472/AFJBS.6.9.2024.281-288

African Journal of Biological Sciences

Journal homepage: http://www.afjbs.com

Research Paper Open Access

CODE SCENTS' EFFECTS ON VARIOUS PROJECT VERSIONS

[1]Mr. Amandeep Singh Arora
[1]Associate Professor, Don Bosco Institute of Technology
[2]Dr. Shalu Tondon
[2]Associate Professor, Don Bosco Institute of Technology
[3]Ms. Charanpreet Kaur
[3]Associate Professor, Don Bosco Institute of Technology
[4]Ms. Prabhjot Kaur
[4]Assistant Professor, SGTBIMIT
[5]Ms. Harleen Kaur
[5]Assistant Professor, SGTBIMIT
[6]Dr. Vikas Rao Vadi
[6]Professor, Don Bosco Institute of Technology

Article History
Volume 6,Issue 9, 2024
Received:14Mar 2024
Accepted : 09 Apr 2024
doi: 10.33472/AFJBS.6.9.2024.281-288

ABSTRACT
This work aims to investigate the effects of code smells on several iterations of three projects: ant, oryx, and mct. The projects and their many iterations are included in the data collection that we have utilized. Robusta is the technique used to identify code smells. This program was used to record the values of the various sorts of code smell that were found in every project version. The Jhawk tool is used to obtain various design metrics' values. Ultimately, we were able to determine the relative criticality of code smells on each project version by comparing the values of the detected smells with metrics.

Index Terms— Code Smells, Detection, Types, Tools, Maintainability

I. INTRODUCTION

Nowadays, with the growth of software practice, the main challenge to deal with, is to deliver a quality software [1], [2]. Thus to improve and deliver a quality software, different useful solutions like code review, refactoring and testing are practiced by the people in the industry [3].

During this phase, the software may encounter some design flaws, that are referred to as "code smells" or can also be termed as "bad smells" [3].Although these code smells are not considered as a fault, they may introduce some kind of bug in the system, which causes declining of the software quality

[4].

But these bad smells can be handled using a suitable software refactoring approach [5], [6].
But another difficulty is faced, that is the tool support whichis critical for software refactoring [7],[8]. The existing smelldetection and refactoring tools that are available are human-driven. They do not take effect until the developers realize, that they should refactor. But the developers can fail to invoke, the tools as frequently as they should [9].

Thus the result is that human-driven refactoring and smell detection tools, fail to drive the developer to first detect andthen resolve code smells [10].

In the research performed, a code smell tool Robusta is used, to detect code smells in three versions of three projects that are ant [14], oryx [15] and mct [16] respectively. Then a tool jhawk is used to determine the metrics of those three versions of the projects. With help of these tools and project versions, the relative weights ofevery metrics for a given code smell is calculated. Thus, in the end it is shown that the relative criticality of a code smell and the contribution by the metrics is same for all the software versions released. Thus through this research, it is shown that in every version of the projects, the changes were done to improve the design flaws in the project that affect its maintenance very minimal. Weights are assigned to every code smell and metrics, and with the help of making a pairwise comparison, the impact of code smells onthe projects is determined [3]. The code smells are produced due to the presence of design flaws, therefore mapping the code smells to the design metrics help the software developer to measure the impact of code smells onthe software in a better way [3].

II. LITERATURE REVIEW

Code smell is a design flaw which one of the main reasons which cause the software system to lose its quality traits. Code smells were discovered and introduced by Fowler andBeck [5].
Thus the presence of a code smell makes the software system difficult to change. Therefore, proper refactoring techniques are used to resolve the flaws produced by code smells. But if the refactoring techniques are not appropriate they may also introduce some more faults into the system [3].
Also, some of the studies have established a relationship between code smells and design metrics [3] [7].
In our study we focus on mainly three code smells which arethe following:-

1. **Empty Catch Block:**

Empty catch blocks are referred to as a type of a code smell in most languages. The main idea here is that one uses exceptions for exceptional situations and not to use them for any type of logical control. All exceptions must behandled somewhere [12].

For example:try {
}
Some Object.something();

The approach that is used to express the opinion is pairwise comparison [8]. In the research performed different code smells namely, empty catch block, unprotected main program and nested try statement are used as criteria.

```
catch(Exception e) {
// should never happen
}
```

Thus the case should be, that either code can handlethe exception, then the catch clause

shouldn't be empty, or the code cannot handle theexception, then there should not be a catch block at all [11].

2. Unprotected Main Program:

Checked exceptions need to be declared in a methodor constructor's throws clause if, they can be thrown by the execution of the method orconstructor and they propagate outside the method or constructor boundary.For example:

```
public static void main(String[] args) throws Exception {
// this should be avoided
```

3. Nested try statement:

Now, this practice that is followed by most of the software developers does, reduces the code-readability and thus, should be avoided. A developer must find an easierand cleaner way to deal with this kinda code smell byadding a finally construct.

For example:

```
Try{
…code} Catch(FirstException e){
….. do something} Catch(SecodException e){
…..do something else}
```

A programmer should always look for a chance forcombining nested try statements [13].

III. METHODOLOGY

The goal of a software maintenance team is that it designs a good and effective refactoring strategy, which will reduce the flaws that are caused because of the presence of code smells in a software system. The strategy chosen should do this with minimum change in the source code and alsomaintain the code's consistency [3].

As mentioned, code smells present in the system are flawsin the source code that can affect the development of a software and also make the system complex and inflexible. Now the question arises, how to select an appropriate refactoring strategy. This can be achieved by measuring the presence of code smells and the damage they can cause to the software.

The aim of the study is to first obtain the weight of code smells by measuring their presence with the help of Robusta tool for three versions released for all three chosen projects that are, ant, oryx and mct.

Since code smells do arise because of the presence of design flaws, therefore there exists a co-relationship between code smells and design metrics. Thus when the hierarchy is extended one more level towards the design metrics, it gives a better insight into the system.

Methodology adopted for the research is as follows:-

STEP 1-

Code smells consider for this study are empty catch block, unprotected main program and nested try statement. These code smells are taken as criteria and will help in determining the level of refactoring i.e. required in the software. Code smells and metrics considered are shown in table 1.

S.NO.	CODE SMELL AND CRITERIA NAME	METRICS NAME	METRICS DESCRIPTION
1	EMPTY CATCHBLOCK	M1	TOTAL NO. COMMENT LINES
2	UNPROTECTED MAIN PROGRAM	M2	TOTAL NO.OF JAVA STATEMENTS
3	NESTED TRY STATEMENT	M3	TOTAL LINES OF CODE

Table 1: Code smells and metrics used

STEP 2-

In this step, a hierarchy is shown to represent the code smells and their relationship with the chosen metrics. The result is shown in Figure 1. By arranging the criteria and their criteria in a hierarchal structure will give the decision maker an opportunity to focus better on criteria and their dependence on sub-criteria [3].

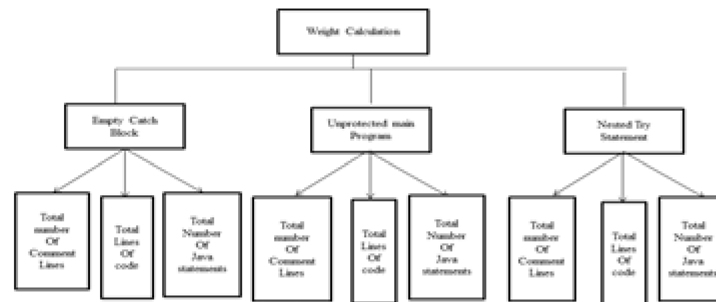

Fig 1: Hierarchal Structure of code smells and metrics

STEP 3-

The weights of each code smell are calculated using comparison between the code smell values measured by ROBUSTA and three versions released of Ant project. The normalized value is obtained by taking each value and dividing it by the summation of its column values. The weights are computed by taking the summation of row value and then dividing each it by total no. of criteriaconsidered.[3] The normalized value and weights of code smells arecalculated and shown in table 2.

Input				Normalized Value			Weights
	C1	C2	C3				
P1	2	4	8	0.33	0.17	0.42	0.307
P2	2	4	6	0.33	0.17	0.31	0.27
P3	2	16	5	0.33	0.17	0.26	0.25
sum	6	24	19				

Table 2: Pairwise comparison and normalized value of codesmell for ant

STEP 4-

As shown in figure 1, each code smell is dependentof 3 design metrics. The weight of each metrics is calculatedby considering the pair wise comparison metrics as done in step 3. The Input values of this matrix is calculated byJHAWK tool. Table 3 shows the weights calculated.

Input				Normali-zed	Valu-e		Weights
	M1	M2	M3				
P1	502	390	278	0.33	0.33	0.33	0.33
P2	502	390	278	0.33	0.33	0.33	0.33
P3	502	390	278	0.33	0.33	0.33	0.33
sum	1506	1170	834				

Table 3: Pair wise comparison and normalised weight of metrics for ant

STEP 5-

In this final step, relative weights of each metrics of a code smell is calculated by,

Rwi = Cwi * Mwi

where, Rwi= relative weight of each metrics of a code smell, Cwi= weight of code smell, Mwi= weight of metrics The weights calculated are shown in table 4. The weights calculated are global weights that reflect the relative criticality of a code smell on the ant software system.

Code Smell	Weight (Cwi)	Metrics Weight (Mwi)		Relative weight (Rwi)
C1	0.307	M1	0.33	0.10131
		M2	0.33	0.10131
		M2	0.33	0.10131
C2	0.27	M1	0.33	0.0891
		M2	0.33	0.0891
		M3	0.33	0.0891
C3	0.25	M1	0.33	0.0825
		M2	0.33	0.0825
		M3	0.33	0.0825

Table 4: Relative Weights of metrics and relative impact of code smell for ant

The same steps, from step 1 to step 5 are performed for two more projects namely, oryx and mct to validate the results. The code smell and metrics criteria chosen for all three projects are same. After performing the steps for oryx and mct project the following results are obtained:

The normalized values and weights of code smell calculated for oryx project are shown in table 5.

Input				Normalize Value-d			Weights
	C1	C2	C3				
P1	20	21	16	0.33	0.32	0.32	0.32
P2	20	22	15	0.33	0.32	0.31	0.32
P3	20	22	18	0.33	0.32	0.36	0.34
sum	60	65	49				

Table 5: Pairwise comparison and normalized value of code smell for oryx

Table 6 shows the normalized value and weights calculated of metrics for oryx :

Input				Normali -zed	Valu -e		Weights
	M1	M2	M3				
P1	158	169	196	0.33	0.34	0.34	0.33
P2	158	168	195	0.33	0.34	0.34	0.33
P3	155	162	188	0.33	0.34	0.32	0.99
sum	471	499	834				

Table 6: Pair wise comparison and normalized weight of metrics for oryx

The weights calculated are shown in table 7. The weights calculated are global weights that reflect the relative criticality of a code smell on the oryx software system.

Code Smell	Weight (Cwi)	Metrics Weight (Mwi)		Relative weight (Rwi)
C1	0.32	M1	0.33	0.1056
		M2	0.33	0.1056
		M2	0.99	0.3168
C2	0.32	M1	0.33	0.1056
		M2	0.33	0.1056
		M3	0.99	0.3168
C3	0.34	M1	0.33	0.112
		M2	0.33	0.112

Table 7: Relative Weights of metrics and relative impact of code smell for oryx The normalized value and weights of code smells calculated for mct project are shown in table 8.

Input				Normaliz-ed	Val-ue		Weights
	C1	C2	C3				
P1	93	25	27	0.24	0.32	0.28	0.28
P2	143	25	34	0.34	0.33	0.36	0.35
P3	147	26	33	0.38	0.33	0.35	0.35
sum	383	77	94				

Table 8: Pair wise comparison and normalized value of code smell for mct

Table 9 shows the normalised value and weights calculated of metrics for mct:

Input				Normalized	Value		Weights
	M1	M2	M3				

P1	151	279	364	0.33	0.33	0.33	0.33
P2	151	279	364	0.33	0.33	0.33	0.33
P3	151	279	364	0.33	0.33	0.33	0.33
sum	453	837	1092				

Table 9: Pair wise comparison and normalised weight of metrics for mct

The weights calculated are shown in table 10. The weights calculated are global weights that reflect the relative criticality of a code smell on the mct software system.

CodeSmell	Weight(Cwi)	MetricsWeight (Mwi)		Relative weight (Rwi)
C1	0.28	M1	0.33	0.0924
		M2	0.33	0.0924
		M2	0.33	0.0924
C2	0.35	M1	0.33	0.1155
		M2	0.33	0.1155
		M3	0.33	0.1155
C3	0.35	M1	0.33	0.1155
		M2	0.33	0.1155
		M3	0.33	0.1155

Table 10: Relative Weights of metrics and relative impactof code smell for mct

IV. RESULT

Three medium sized software is considered for the study and to understand the impact of code smells on each versionof the project. Its observed that three versions of the three software are released, and for all the tree versions the relative critically of the code smell is same. Thus, in every release, there was no improvement at the design level which, thus makes the project more complex, decreases its maintainability and the development of software is affected.

V. CONCLUSION AND FUTURE SCOPE

Thus, as observed for every version of the software, the relative impact of code smell is same. Thus, the development team for a software system, should focus on design flows of the system to increase its shelf life. Less thenumber of code smells, less will be their impact on the system. Resulting in a more maintainable system. Further, more software system can be analyzed. Thus, the development team should make sure to also focus on the design of the system. Every release proposed for a software system should also be improved at design level.

REFERENCES

1. Kapur PK, Singh VB, Anand S, Yadavalli VS "Software reliability growth model with change point and effort control using a power function of the testing time" Int JProd Res 46(3):771-787,

2008.
2. Kapur PK, Nagpal S, Khatri SK, Basirzadeh M "Enhancing software reliability of a complex softwaresystem architecture using artificial neural-networks ensemble" Int J Reliab Qual Saf Eng 18(03):271–284, 2011.
3. Rajni Sehgal, Deepti Mehrotra, Manju Bala "Analysis ofcode smell to quantify the refactoring" Int J Syst Assur Eng Manag, 2017.
4. D'Ambros M, Bacchelli A, Lanza M "On the impact of design flaws on software defects" In 10th International conference on quality software (QSIC), IEEE, pp 23–31, 2010.
5. M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts "Refactoring: Improving the Design of Existing Code" Addison Wesley Professional, 1999.
6. W.C. Wake, Refactoring Workbook. Addison Wesley, Aug. 2003.
7. T. Mens and T. Touwe, "A Survey of SoftwareRefactoring," IEEE Trans. Software Eng., vol. 30, no. 2, pp. 126-139, Feb. 2004.
8. E. Mealy and P. Strooper, "Evaluating Software Refactoring Tool Support," Proc. Australian Software Eng. Conf., p. 10, Apr. 2006.
9. E. Murphy-Hill, C. Parnin, and A.P. Black, "How We Refactor, and How We Know It" IEEE Trans. Software Eng., vol. 38, no. 1 ,pp. 5-18, Jan./Feb. 2012.
10. Hui Liu, Xue Guo, and Weizhong Shao, "Monitor-Based Instant Software Refactoring" Ieee Trans. Software Eng., vol. 39, no. 8, August 2013.
11. wiki.c2.com/?EmptyCatchClause
12. softwareengineering.stackexchange.com/questions/1680 7/is-it-ever-ok-to-have-an-empty-catch statement
13. softwareengineering.stackexchange.com/questions/118788/is-using-nested-try-catch- blocks-an-anti-pattern
14. https://github.com/apache/ant
15. https://github.com/OryxProject/oryx
16. https://github.com/nasa/mct