ISSN 1846-6168 (Print), ISSN 1848-5588 (Online)
https://doi.org/10.31803/tg-20210207102610

Preliminary communication

Design and Implementation of a Web-Based Application for Code Smells Repository

Lida Bamizadeh*, Binod Kumar, Ajay Kumar, Shailaja Shirwaikar

Abstract: Pitfalls in software development process can be prevented by learning from other people's mistakes. Software practitioners and researchers document lessons learned and the knowledge about best practices is spread over literature. Presence of code smells does not indicate that software won't work, but it will reveal deeper problems and rising risk of failure in future. Software metrics are applied to detect code smells whereas refactoring can remove code smells, improve code quality and make it simpler and cleaner. Detection tools facilitate management of code smells. Knowledge about code smells and related concepts can assist the software maintenance process. Exploratory analysis of code smells carried out in this paper, covers collecting data about code smells, identifying related concepts, categorizing and organizing this knowledge into a code smell repository, which can be made available to software developers. A detailed literature survey is carried out to identify code smells and related concepts. An initial list of 22 code smells proposed in 1999 has grown over the years into 65 code smells. The relationship between code smells, software metrics, refactoring methods and detection tools available in literature is also documented. Templates are designed that capture knowledge about code smells and related concepts. A code smell repository is designed and implemented to maintain all the information gathered about code smells and related concepts and is made available to software practitioners. All the knowledge about code smells found in literature is collected, organized and made accessible.

Keywords: code repository; code smell; detection tool; refactoring; software metric

1 INTRODUCTION

Software quality goes to ruin over time because of various reasons such as software ageing, improper design, unsuitable requirement analysis and inappropriate coding practices. Code smell is an indication of some obstacles in the code that shows something is wrong in some parts of code or system design [1]. Bad smell occurrence has a drastic influence on the quality of code. It makes system more complex, less comprehensible and causes maintainability problems [2, 3]. Bad smells are usually not bugs; however, researchers have proposed that a huge number of bad smells connect with bugs and maintainability issues [4-7]. Bad smells don't currently inhibit the functioning of code. But, they detect clear signs in design which may lead to slowing down development or growing the probability of bugs or software rot because of long term decays. In 1990, Kent Beck proposed that refactoring can modify source code to improve its quality. Refactoring is a systematic process of improving source code without creating new functionality that can change a disorder into spotless code and uncomplicated design [8]. Code smell detection can be effectively carried out by using appropriate software metrics [9]. Software metric is a measurement indicator for the latent attributes possessed by software system or software development process [4, 10, 11]. These detection approaches interpret code metrics which are evoked from a particular system element by applying a set of threshold filter rules [12]. The main target of this strategy is providing a mechanism for engineers that give permission to them to work on a more abstract level that conceptually is closer to real goals in using metrics. Furthermore, several tools have been developed for detection of code smells and improve the code quality during software development [13]. Software tools support developers by automatic or semi-automatic detection of bad smells. Tools focus on the entities which most likely present code smells [14].

This paper proposes an exploratory study of code smells. Literature survey shows that there are plenty of code smells with corresponding detection methods using software metrics. Also, there is a large set of tools that support code smells detection. Moreover, there are well defined Refactoring methods that can be used to remove code smells. There is a need to organize this knowledge into a Code smells repository so that it is readily available to developers and practitioners.

The contribution of this paper is organized as follows: section 2 describes background and related work. Exploratory analysis of code smells is explained in section 3. Organizing the code smell knowledge is presented in section 4 which is followed by conclusion in section 5.

2 BACKGROUND AND RELATED WORKS

Several scientists such as Opdyke et al. [15] showed that some situations in source code may need refactoring. The process of changing a software system in this manner that external behaviour does not change but improves its internal structure is called refactoring. It can improve the design of a software process and reduce its complexity. After refactoring, software systems are easier to comprehend and maintain. Webster [16] and Brown et al. [17] discovered some code smells such as Blob, Spaghetti Code, etc. [18]. Later, Kent Beck and Martin Fowler [19] called those situations, which may need refactoring as the bad smells. An initial list of code smells was proposed by them which was indicative of something incorrect in the system code. They introduced a list of 22 code smells without categorizing them and claimed that there is not a set of precise metrics which can be specified to recognise the need of refactoring. Thus, bad smells are kind of cooperation amongst the ambiguous programming and precise source code metrics. Fowler presented a group of refactoring with step wise comments on how each smell can be removed. He did not give the particular characteristics, detecting techniques and refactoring process. Van Emden and Moonen [20] proposed the first formalization of code smells. They revealed the undesirable effect of bad smells on the software product.

They suggested an automatic detection and visualization of code smell, with a methodology for reducing the impact of code smells on java source code. "Jcosmo" was the name of resulted work in code smell browser. Later, they had other survey and discovered that presence of smells has maximum influence on quality of software [18]. Kerievsky [21] presented more refactoring. Also, he introduced some new code smells such as Conditional Complexity, Combinatorial Explosion and Indecent Exposure in his refactoring book. Mantyla [22] presented Divergent Change as concealed smell. This smell cannot be detected by a simple look at the code or by tools. Also, detecting process need good understanding of the code and having experience for implementing the changes to the source code. Then, in 2003, he [23] introduced more smells. In addition, he discovered a classification of 22 code smells into seven units where every individual unit reveals a similar impression [23, 13]. Li and Shatnawi [24] examined the association among class error probability and code smells for three different levels such as High (Blocker and Critical), Medium (Major), and Low (Normal and Minor). They described that refactoring of a class improves the architectural quality as well as decreases the probability of the class errors when system is released. Also, in 2008 they [25] extended their study about relationship between software metrics, code smells and class error probability. Fontana et al. [26] proposed a comparative study of code smells which are detected by various refactoring tools and their support of semi-automatic refactoring. Ouni et al. [27] defined a search based refactoring strategy for maintaining domain semantic of a code when refactoring is decided/ implemented automatically. They discussed that refactoring may be syntactically correct and have right behaviour but model incorrectly the domain semantics. Palomba et al. [28] surveyed observations of developers about bad smells. They mentioned that there is a gap between theory and practice. Their survey promised insights about bad smells which are not yet explored sufficiently. Pinto and Kamei [29] examined StackOverflow's data for exploring obstacles for approval of code smell detection tools. They prepared a list of problems that revealed the adoption/usability problems, which users explained about StackOverflow. Tufano et al. [30] surveyed hundreds of projects to explore the problems of bad smells. They discovered the reason for bad smells in the code. Kaur and Dhiman [31] had a detailed survey on Search-Based Tools and Techniques to Identify Bad Code Smells in Object-Oriented Systems. Authors point out lack of a standard benchmark system for comparing outcomes of existing's code smell detection strategies. Fontana et al. [32] believed that code smells and architectural smells are not same. They suggested developers to more focus on hazardous architectural smells. Reis et al. [33] performed a Systematic Literature Review (SLR) on the state-of-the-art methods and tools applied for code smells detection and visualization. Their results showed that the most repeatedly applied detection methods are based on search-based techniques, which mainly apply ML algorithms. Martins et al. [34] presented a survey on harmfulness of co-occurrences of code smells and its influences on Internal Quality Attributes. The elimination of code smells co-occurrences reduce complexity of the system. Kaur [35] published a Systematic Literature Review on Empirical Analysis of the Relationship between Code Smells and Software Quality Attributes. Researcher observed that most used data sets for studies are small in size and written in Java programming language. Also, most impact of code smells is on external quality attributes. Al-Shaaby et al. [36] recently presented a systematic literature review with reference to bad smell detection using machine learning techniques. Their research outcomes showed that God Class and Long Method, Feature Envy, and Data Class are the most occurring detected code smells and Java programing and Weka have most used by researchers.

3 EXPLORATORY ANALYSIS OF CODE SMELLS

Exploratory Analysis of code smells involves collecting data about code smells, identifying related concepts, categorizing and organizing this knowledge into a code smell repository so that it can be made readily available to software developers and practitioners.

3.1 Collection of Data about Code Smells

Kent Beck as the originator of extreme programming revealed the importance of design quality through the developing software in 1990s and made popular the usage of word code smell. This word grew into a universal term in coding when it was introduced in the book Refactoring: Improving the Design of Existing Code by Martin Fowler, a famous software scientist which propagated the practice of refactoring. An initial list of 22 code smells was introduced by Kent Beck and Martin Fowler in 1999 as situations revealing of something improper in the system code. Initial list has grown over the years and knowledge about a large set of code smells is spread out across the literature. For the exploratory study, 65 code smells are gathered from the existing literature as shown in Tab. 1. Name of Code smells are a part of designer's language vocabulary. Sometimes newbies designers don't certainly know code smell's particular definitions and simply use them out of familiarity. Researcher has prepared uncomplicated short definitions via Tab. 1 to assist designers and other researchers to identify promising motivations for solving code smell problems.

Table 1 Code Smells and Their Brief Definitions

No	Name	Definition
1	Duplicate Code	Presence of same code structure at more than one place.
2	Long Method (Function)	Method is too long in the sense of the functionalities executed by it.
3	Large Class	Class has large number of instance variables and attempts to organize many works.
4	Long Parameter List	List of parameters is very lengthy.
5	Divergent Change	Single class requires many changes in the code for the various objectives.
6	Shotgun Surgery	Single change applies to several different classes of code simultaneously.
7	Feature Envy	Method looks to be more concerned in other class than it is real occupied.

8	Data Clumps	Same set of data items often appear together in different places.
9	Primitive Obsession	Instead of small objects, extreme use of primitive data types.
10	Switch Statements	Equal switch statements scattered across the code in the several places.
11	Lazy Element	An element which is not doing sufficient work.
12	Lazy Class (Freeloader)	A class which is not doing sufficient work.
13	Data Class	Class is container for data used by other classes and cannot work independently in own data.
14	Excessively Large (long) Identifier	Identifier length is very large and makes disambiguation in the software architecture.
15	Excessively Small (short) Identifier	Identifier length is very small and does not reveal its function obviously.
16	Contrived complexity	Design pattern is overcomplicated where a simpler design could be used.
17	Complex conditionals	Large conditional logic blocks, especially blocks that tend to grow larger or change considerably over time.
18	Temporary fields	They acquire their values, are required by objects under certain situations, and are empty outside of these situations.
19	Refused Bequest	Child class does not use derivative functionality of the superclass to happen inheritance rejection.
20	Middle man	When a class performs only delegating work to another class.
21	God (Blob) Class	Class has tendency to localize the intelligence of the system and trying to do much.
22	Alternative Classes with Different Interfaces	Two different classes perform similar functions with different method signatures.
23	Parallel Inheritance Hierarchies	Two parallel class hierarchies stand and each of these hierarchies must to be extended.
24	Message Chains	When in the code, there are a series of calls resembling a->b( )->c( )->d( )
25	Comments	When a method is full of descriptive comments.
26	Dead Code	The code which has been used earlier, but is not presently used.
27	Brain Class	Class centralize system functionality but does not use considerable data of foreign classes and is more cohesive.
28	Brain Method	Brain Methods centralize the functionality of a class.
29	Extensive (Dispersed) Coupling	A single operation calls one or few methods from extreme number of provider classes.
30	Intensive Coupling	A method calls many other operations in the system from one or a few classes.
31	Tradition Breaker	Inherited class hardly concentrates inherited services which are unrelated on inherited functionality by base class.
32	Spaghetti Code	A class without structure executes long and complex methods, connects among them without parameters using global variables.
33	Speculative Generality	An abstract class that is unused, but will be used in the system in coming system releases.
34	Inappropriate Intimacy	Two classes exhibiting high coupling between them or one class consumes the internal fields and methods of another class.
35	Complex Class	Classes having high complexity.
36	Class Data Should Be Private (CDSBP)	A class exposes its attributes and violates the principle of data hiding.
37	Instanceof	Having a chain of "instanceof" operators in the same block of code.
38	Typecast	The process of explicitly converting an object from one class type into another.
39	Missing Template Method	Two different components have major similarities, but do not use an interface.
40	Cyclic (Circular) Dependencies	Two or more subsystems are involved in one cycle and this is contravention of Acyclic Dependencies Principle.
41	Blob Operation	Huge and complex operation have a tendency to centralize too much of the functionality of a class or module.
42	Sibling Duplication	An equivalent functionality described by two or more siblings in an inheritance hierarchy.
43	Internal Duplication	Duplication among portions of the one class or module.
44	External Duplication	Duplication among unrelated capsules of the system.
45	Distorted Hierarchy	Uncommonly narrow and deep Inheritance hierarchy. A popular value for this depth is six.
46	Unstable Dependencies	Dependencies between subsystems in a design are not in direction of the stability of subsystems.
47	Schizophrenic Class	A class includes separate sets of public methods that are used by separate groups of client classes.
48	Incomplete Class Library	Library hasn't prepared the features or has declined to implement them.
49	Stable Abstraction Breaker (SAPBreakers)	A subsystem is not as abstract as it is stable.
50	Mysterious (Uncommunicative) Name	A mysterious name of functions, modules, variables and classes does not lead into its intent well enough.
51	Mutable data	Unexpected consequences and bugs can be produce after changing the data.
52	Global data	Everyone from everywhere can modify global data and this is a problem of it.
53	Similar subclasses	There is a bunch of almost similar subclasses of a class.
54	Inconsistent Names	A group of standard terminology should choose and then, use it among the methods. Example: Open () ► Close ().
55	Combinatorial Explosion	Lots of code does almost the same thing, but with tiny variations in data or behaviour.
56	Type Embedded in Name	Placing types in method names.
57	Oddball Solution	Two similar problems in one system, and one problem solves one way and other solves other way.
58	Indecent Exposure	When methods or classes unnecessarily expose their internals.
59	Solution Sprawl	Code and/or data used in execution of a responsibility becomes sprawled through several classes.
60	Excessive return of data	A function or method which returns more than what each of its callers requires.
61	Excessively long (God) line of code	When a line of code is too long.
62	Functional Decomposition	Writing highly procedural and non-object oriented code in an object oriented language. It happens while a class is considered with the intent of performing a single function.
63	Deficient encapsulation	Declare availability of one or more members of an abstraction is more permissive than actually necessary.
64	God Method	Method gets more functionality until it becomes out of control and difficult to maintain and extend.
65	Type checking	Presented for Selecting a variation of an algorithm that should be executed based on the value of an attribute.

3.2 Identifying Related Concepts

Code smells are some symptoms in the source code that probably indicates a deeper problem in software system. Detection of code smells is challenging for practitioners and developers. Different viewpoints conduct to the application of several detection metrics, detection tools and refactoring actions [14]. Software metrics are a standard measurement by using performance value named threshold to assess the maintainability of the software systems and to distinguish code smells. Tools are another way of code smells detection. A variety of detection tools have been developed for detection of bad smells based on different approaches and specific parameters for detecting particular smells. Refactoring is an organized procedure for improving source code without making new functionalities that change code to clean code with a simple design. Fig. 1 depicts that code smells are the centre of study along with the related concepts that are software metrics, detection tools and refactoring actions.

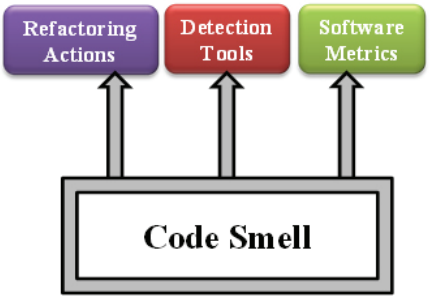

Figure 1 Concepts related to code smells

Following subsections are used for describing these related concepts in further detail.

3.2.1 Detection Tools

There are several tools and IDE (integrated development environment) available for detecting of code smells [37]. Code smells are suggested as an attempt by programmers to reform their software. When programmers are writing their code, bad smells go unnoticed. Therefore, detection tools are developed to make programmers aware about the existence of bad smells in their code and to aid them recognize the reason of those bad smells. Several code smell detection tools are available but it is difficult to enumerate all of them and define exactly which bad smells they are able to detect. Therefore, a short introduction is provided to some of the well-known tools.

a) **infusion**: inFusion is the modern and commercial development of iPlasma and detects 22 code smells. Refactoring is not available but it is linked to the code. [13, 14, 26, 38]

b) **iPlasma**: iPlasma is for quality assessment of object-oriented systems, supports all steps of analysis. Refactoring and link to code are not available. [26, 38]

c) **JDeodorant**: JDeodorant automatically recognizes code smells and is able to determine proper sequence of refactoring. Also, it is linked to code. [11, 13, 14, 38]

d) **JSpIRIT**: JSpIRIT supports java codes to recognize and arrange code smells. Automated refactoring and link to code are not available. [14, 26, 38, 39]

e) **PMD**: PMD supports programs and searches for faults. Refactoring is not available and detection technique is based on software metrics. [13, 14, 26, 38]

f) **Checkstyle**: Checkstyle is similar to PMD for using software metrics and thresholds for detection of bad smells. Automatic refactoring is not available and it is linked to the code. [13, 26, 38]

g) **Stench Blossom**: Stench Blossom gives a visualization environment to show the programmers a high-level outlook of the bad smells in their code. Automated refactoring is not available but there is direct link to code. [13, 26, 38]

h) **DÉCOR**: DÉCOR automatically permits the specification and detection of bad smells. Refactoring is available as well as code links. [13, 26, 38]

i) **inCode**: inCode is commercial and based on inFusion for detecting of bad smells that supports programmers for writing code in programming environment. [13, 40]

Table 2 Code Smells and Detection Tools That Detect Them

No	Code Smell	Tools								
		Checkstyle	DÉCOR	inFusion	iPlasma	JDeodorant	PMD	Stench Blossom	Jspirit	inCode
1	Brain Class			✓	✓					
2	Brain Method			✓	✓				✓	
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
28	Dispersed Coupling				✓				✓	

Tab. 2 shows relationship between code smells and detection tools. According to the table, iPlasma scores maximum points on detection of code smells with 17 detected code smells. It can be interpreted that iPlasma is a functional tool compared to other tools and should be selected by developers as it covers the detection of larger set of code smells. There are only 28 code smells that are detected by one or other of these 9 detection tools.

3.2.2 Software Metrics

Software Quality Metrics refer to measurement of software attributes related to software quality during software development process. Many software metrics are available to systems realized in various paradigms like Objects Oriented Programming (OOP). Finding factors of software quality and planning them into quantitative measures is a critical issue in sustainable success of an end product. Software metric has involved a lot of consideration between researchers and developers in last one decade [41]. Computer science experts are placing all their struggles in measuring quantitative information from software component. Therefore, software metrics are often classified into some types [42]. It is depending on different lookouts. Shepperd and Ince [43] proposed a classification of two metrics: traditional metrics and object-oriented metrics.

Later, Saker [44] suggested a category of software metrics established upon subject and paradigm. In his category software metrics divided to project based metrics and design based metrics. Fenton and Bieman [45] offered different category that it was two dimensional classifications and divided to project metrics (product, process or resources) and the level of visibility that can be internal or external metrics [42]. Also, by other researchers it was divided into basic and additional metrics, objective or subjective, project classification, and static and dynamic. For assessment of quality of software systems, it is significant to define thresholds for software metrics [46]. Software metrics are deliberated for bad smell detection in source code. Existing bad smells in source code shows inacceptable architecture design of software that makes it severe to maintain in future. Software measurement is a process that represents software product or process characteristic to a numeric value [45]. The results are compared with a set of standards that are defined by individuals or organizations and a software quality is concluded [47]. Software metrics can be used to each phase of software development process such as requirements, design, implementation, testing and evaluation, maintenance and use for evaluating of quality of software. Tab. 3 shows 49 software metrics used for detection of code smells with abbreviation and a short definition. All code smells in Tab. 1 do not have metrics to detect.

Table 3 Software Metrics, Their Abbreviations and Brief Definitions

No	Metrics	Abbreviation	Definition
1	Number of Lines of code	LOC	Counting of lines of source code.
2	McCabe Cyclomatic Complexity per module	VG	It measures complexity of source code using number of linearly independent paths of a program.
⋮	⋮	⋮	⋮
49	Number of concerns per component	NCC	Number of concerns per component

Each code smell can be detected by one or more metrics. For instance, Feature Envy can be detected by three metrics [13].

In programming, objects are used as a structure for keeping together data and operations which process that data. Feature Envy indicates Methods that look to be more concerned in other classes than its real occupied. Feature Envy methods access a variety of data of foreign classes. This may possibly is because of misplacing methods and they should move to another class. Data and operations should be close as feasible. This proximity can help to improve the cohesion and ripple effects reduction. Detection of Feature Envy considers counting the number of data members that used by method outside of its own class. Detection technique follows below steps:

1) Method uses more than few attributes of other classes and this measures by *ATFD* (Access To Foreign Data) metric.
2) Method uses more attributes from other classes compare as its own class and this measure by *LAA* (Locality of Attribute Accesses) metric.
3) The used foreign attributes are from a few outside classes and this measure by *FDP* (Foreign Data Providers) metric. This step considers because, if method uses foreign attributes of one or two outside classes it is feature envy smell but if method uses foreign attributes of more outside classes this is Brain Class smell. Therefore, for separation of this two smells researchers consider third condition.

In additional, researchers consider counting of all dependencies of the method, either inside its own class or outside its own class, and they use *FDP* metric because if method uses a few attributes from foreign classes, method can move easily to foreign classes and dispersion of classes will decrease. Also, foreign class includes less functionality and Feature Envy method has high complexity and size.

Feature Envy can be detected by the Eq. (1) and Fig. 2.

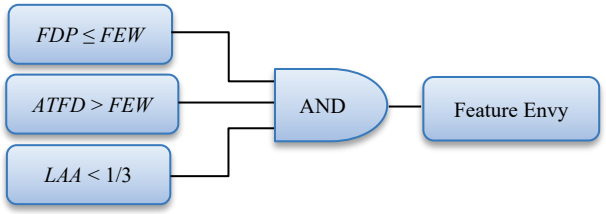

Figure 2 Detection technique for Feature Envy [13, 48].

$$FDP \le FEW \wedge ATFD > FEW \wedge LAA < \frac{1}{3} \quad (1)$$

where *FEW* takes the value of 5 [13, 48].

On the other hand, Large Class can be detected only by LOC [13].

Tab. 4 illustrates the relationship between some code smells and their code smell detection metrics.

Table 4 Code Smells and Software Metrics Used in Detection Them

No	Code Smells						
	No	1	…	19	20	21	22
	Metrics	God Class	…	God Method	Inappropriate Intimacy	Divergent change	Shotgun surgery
1	LOC	✓	…				
2	VG		…	✓			
.	.	…	…	…	…	…	…
36	CM		…			✓	✓
38	NCC		…			✓	

3.2.3 Refactoring Actions

As reported by Fowler, code smells can be removed by refactoring. Refactoring develops the design of existing code of software system by modification of internal structure without affecting its external structure. The main target of refactoring action is improving software design quality and developing quality features like understandability, flexibility, and reusability. Refactoring is not developing the design of the software system through its initial step of design, but

developing its design through the maintenance phase [7]. Tab. 5 shows refactoring actions name and definition. In addition, Tab. 6 describes relationship between code smells and corresponding refactoring actions.

Table 5 List of Refactoring Methods

No	Refactoring Action Name	Definition
1	Add Parameter	If there is not available enough data to execute particular actions for a method, then make a new parameter to pass the essential data.
2	Inline Method	If body of a method is clearer rather than the method itself, then replace calls to the method with the method's content and remove the method itself.
…	…	…
87	Unify Interfaces with Adapter	Clients cooperate with two classes, but one of them has a preferred interface. Then, these interfaces unify with an adapter.

Table 6 Code Smells with Corresponding Refactoring Actions

No	Code Smells	Refactoring Actions
1	Long Method	Extract Method, Replace Temp with Query, Replace Method with Method Object, Substitute Algorithm, Decompose Conditional, Introduce Parameter Object, Preserve Whole Object, Replace Parameter with Explicit Methods, Replace Conditional Logic with Strategy, Replace Conditional Dispatcher with Command, Compose Method, Move Accumulation to Collecting Parameter, Move Accumulation to Visitor
⋮	⋮	⋮
27	Indecent Exposure	Encapsulate Classes with Factory
28	Solution Sprawl	Move Creation Knowledge to Factory

3.3 Categorization of Code Smells and Related Concepts

Categorization is grouping objects according to their similarities and common features or relationship between all members in the group. It is an essential process for cognition of things. Categorization organises knowledge and improves understandability as element inherits categorical attributes. Fig. 3 shows as an overall view of categorization of code smells and related concepts.

Each of these categories is explored further in detail in following subsections.

3.3.1 Categorization of Code Smells

Mantyla [23] proposed a classification of code smells because; some of the code smells are closely related. Each category has an appropriate name which is according to relationship between the bad smells in each category. This classification is provided to better understanding of smells and to identify relationship between them. Over the years, the initial categorization is slightly changed by researchers. As follows in Tab. 7, classification of bad smells with their definition is explained [23, 49, 50].

In literature 22 code smells are classified. Researcher tried to find out the classification of all code smells that are covered in Tab. 1, and Tab. 8 shows categorization of each code smell.

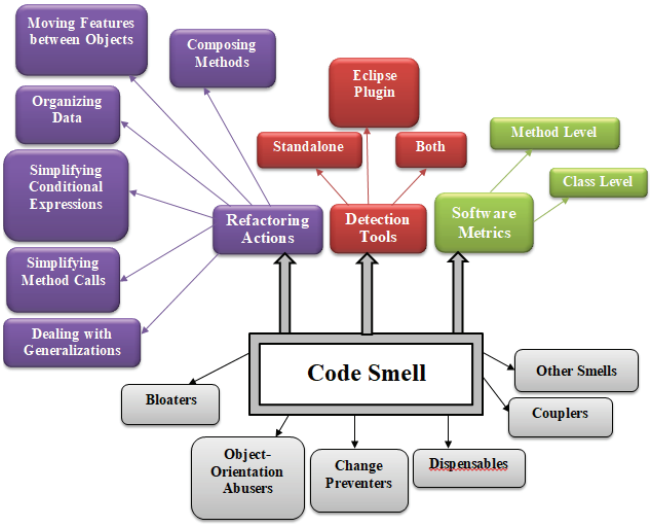

Figure 3 Categorization of code smells and related concepts

Table 7 Code Smells Categories with Their Definitions

No	Category	Definition
1	Bloaters	It reveals one part of code that has grown so large and cannot be successfully handled.
2	Object-Orientation Abusers	It reveals incorrect or incomplete use of object-oriented concepts.
3	Change Preventers	If changing in one place of code requires many changes in other places too.
4	Dispensables	Display something unnecessary in the code whose absence would make the code more effective.
5	Couplers	Lead to excessive coupling among classes or indicate what happens if coupling replaced by excessive delegation.
6	Other Smells	These smells do not fit in any of the above classification.

Table 8 Category-Wise Distribution of Code Smells

No	Classification	Code Smells	Total
1	Bloaters	Long Method, Large Class, Primitive Obsession, Long Parameter List, Data Clumps, Complex conditionals, Blob Operation, Excessively long line of code,	8
2	Object-Orientation Abusers	Switch Statements, Temporary Fields, Refused Bequest, Alternative Classes with Different Interfaces, Parallel Inheritance Hierarchies, God Class, Brain Class, Brain Method, Tradition Breaker, Spaghetti Code, Complex Class, Class Data Should Be Private, Typecast, Cyclic Dependencies, Distorted Hierarchy, Unstable Dependencies, Schizophrenic Class, Stable Abstraction Breaker, Functional Decomposition, God Method, Indecent Exposure, Solution Sprawl, Deficient encapsulation, Type checking	24
3	Change Preventers	Divergent Change, Shotgun Surgery	2
4	Dispensables	Duplicate Code, Lazy Class, Data Class, Dead Code, Speculative Generality, Lazy Element, Contrived complexity, Comments, Instanceof, Sibling Duplication, Internal Duplication, External Duplication, Similar subclasses, Excessive return of data	14
5	Couplers	Feature Envy, Inappropriate Intimacy, Message Chains, Middle Man, Intensive Coupling, Extensive Coupling,	6

6	Other Smells	Missing Template Method, Incomplete Class Library, mysterious name, Mutable data, Global data, Inconsistent Names, Combinatorial Explosion, Type Embedded in Name, Oddball Solution, Excessively Large Identifier, Excessively Small Identifier	11

3.3.2 Categorization of Tools

Various detection tools are able to execute automatic code inspection. Smell detection tools are categorized either as plug-in or as stand-alone application [13]. These tools adopt a little different approaches for detecting code smells. The Eclipse framework is a common integrated advance environment, planned to assist tools that can be used to develop applications and tools or to handle all varieties of documents. A small plug-in loader is placed at the core of Eclipse and entire extra functionalities are performed by plugins [51]. A standalone tool performs locally on the device and doesn't need anything else to be functional. Standalone tools have continuity and interpretation disadvantages. For development of code detection, tool requires a visual integration into the IDE (Integrated development environment). A standalone tool cannot understand which part of code is edited by programmer, therefore continuity cannot be achieved.

Table 9 Detection Tools with Category and Supported Languages

No	Tool	Type	Languages
1	inFusion	Standalone Application	Java, C, C++
2	iPlasma	Standalone Application	C++, Java
3	JDeodorant	Eclipse Plug-in	Java
4	JSpIRIT	Eclipse Plug-in or Standalone Application	Java
5	PMD	Eclipse Plug-in or Standalone Application	Java, C, C++ and others
6	Checkstyle	Eclipse Plug-in	Java
7	Stench Blossom	Eclipse Plug-in	Java
8	DÉCOR	Standalone Application	Java
9	inCode	Eclipse Plug-in or Standalone	Java, C, C++ and

On the other hand, a smell detector plugin shows continuously whether any code smells have been realized without forcing the programmer to leave his IDE. After finding a smell, tool can easily shows the existence of the smell and a suggestion how to remove it. This performance underlines the usability factor. Thus, Smell detection tools with integrated IDEs are more effective compared to stand-alone detection tools. Tab. 9 shows categorization of covered detection tools, and also languages supported by each of them.

3.3.3 Categorization of Software Metric

Every bad smell involves a particular kind of system element like classes or methods which can be appraised by its inner and external characteristics. Metrics can use in file-level, class-level, component-level, method-level, process-level and quantitative values-level metrics [52]. In this exploration study, Software metrics are categorized as class level and method level metrics. Class level metrics measure features of class as well as information on the collaboration among classes. Class level metrics that measure class communications give information for design the system more than code. Some of the class level metrics determine division of labour between methods while others determine the amount of code affect in other classes with changing a special class. The best situation is changes in one class have minimum changes in other classes. When a high level dependency is between classes, they should locate in same package. Method level metrics are one of the most useful metrics. One of the ideal guidelines of programming is that each method should execute a single clear distinct function because a long part of code is difficult to understand [53, 54]. Tab. 10 shows categorization of covered software metrics in Tab. 3.

Table 10 Software Metric's Categorization

No	Metrics Categorization	Software Metrics	Total
1	Class Level	LOC, WMC, DIT, CC, CBO, LCOM, TCC, NOM, NOA, NOC, RFC, NOAM, NBM, WOC, NOPbA, NProtM, BUR, BOvR, AMW, NOPvA, NOProtA, NOPvM, NOPbM, NOProtM, NLOCC, CDE, DAC, LCC, IVMC, NOFF, NOFM, LOMC, NOVC, CHC, DOCM, NCC	36
2	Method Level	VG, PAR, MLOC, LAA, MNL, NOAV, NODM, CP, HM, UP, ATFD, FDP, CM	13

3.3.4 Categorization of Refactoring Actions

Various refactoring actions are available that some researchers have divided them into 6 categories as follows [50].

1) **Composing methods**: Most of the refactoring is concerned with accurately composing methods because extremely long methods are root cause of all destructive qualities. Therefore, this group restructures methods, eliminates code duplication, and provides better future improvements.

2) **Moving features between objects**: Moving functionality between classes, building new classes, and hiding performance features from public access is supported by these refactoring actions.

3) **Organizing data**: This group assists data management, replacing primitives with rich class functionality and helps to solve class associations that construct classes more portable and reusable.

4) **Simplifying conditional expressions**: Preventing conditionals from getting more and more complicated in their logic over time is facilitated by this group of refactoring actions.

5) **Simplifying method calls**: This group streamlines the interfaces for collaboration between classes and creates method calls uncomplicated and more obvious to understand.

6) **Dealing with generalizations**: Moving functionality across the class inheritance hierarchy, building new classes and interfaces, and substituting inheritance with delegation

and vice versa or anything related to abstraction is handled by this group of actions. Researcher tried to classify all refactoring actions in Tab. 5 and shows each code smell belongs to which categorization in Tab. 11.

Table 11 Refactoring Action's Categorization

Categorization of Refactoring	Refactoring Actions	Total
Composing methods	Extract Method, Inline Method, Extract Variable, Inline Temp, Replace Temp with Query, Split Temporary Variable, Remove Assignments to Parameters, Replace Method with Method Object, Substitute Algorithm, Compose Method, Replace Implicit Tree with Composite	11
⋮	⋮	⋮
Dealing with generalizations	Pull Up Field, Pull Up Method, Pull Up Constructor Body, Push Down Field, Push Down Method, Extract Subclass, Extract Superclass, Extract Interface, Collapse Hierarchy, Form Template Method, Replace Inheritance with Delegation, Replace Delegation with Inheritance, Chain Constructors, Extract Composite, Introduce Polymorphic Creation with Factory Method	15

4 ORGANIZING THE CODE SMELL KNOWLEDGE

One of the most important objectives of code smell exploratory study is organisation of knowledge of code smells. Designing of a code smell repository improves software process, decreases the research gaps and prepares structural sources to developers. Organising the code smell knowledge is showed in follows steps.

4.1 Designing Code Smell Template

Designing code smell template is according to relationships between code smells, software metrics, detection tools and refactoring actions. A code smell template is designed and an instance of it is presented.

Code smell Template

Name: name of special code smell
Alias: This is an alternate name for Code Smell
Definition: Definition of special code smell
Links: The list of databases that information about special code smell is available
Category: The name of category that special code smell belongs to
Detection Tools: The name of tools that can detect special code smell
Software Metrics: The name of metrics that can detect special code smell
Refactoring Actions: The name of refactoring techniques that are able to remove special code smell

4.2 Designing Code Smell Database Schema

Based on designed code smell template a schema is designed for describing of its structure. A code smell database schema characterizes the tables and corresponding fields contained in a database. It displays as a list of tables that every table contains a sub list of fields beside the related data type. Code smell database schema includes main tables such as code smell table, metric table, tool table, refactoring table and relational tables between the main tables.

Example of Code smell Template

Name: Lazy Class
Alias: Freeloader
Definition: A class which is not doing sufficient work
Links: https://refactoring.guru/smells/lazy-class, Martin, F. (1999). Refactoring: improving the design of existing code. Pearson Education India.
Category: Dispensables
Detection Tools: DÉCOR, PMD
Software Metrics: LOC, VG, WMC, DIT, CBO, NOM, NOA
Refactoring Actions: Inline Class, Collapse Hierarchy, Inline Singleton

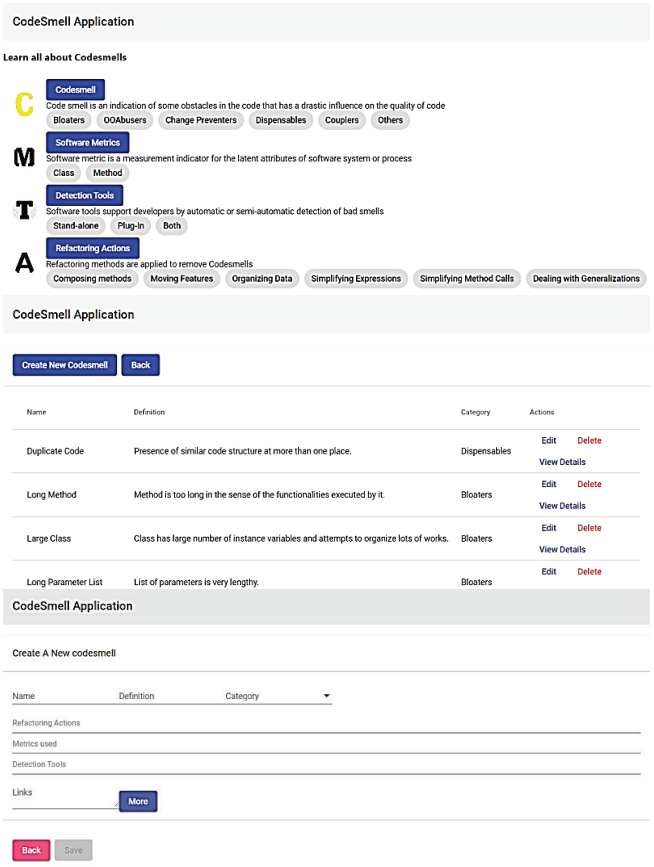

Figure 4 Web page showing Code smell Listing

4.3 Designing Code Smell Database Schema

Based on designed code smell template a schema is designed for describing of its structure. A code smell database schema characterizes the tables and corresponding fields contained in a database. It displays as a list of tables that every table contains a sub list of fields beside the related data type. Code smell database schema includes main tables such as code smell table, metric table, tool table, refactoring table and relational tables between the main tables.

4.4 Making the Knowledge Accessible on Cloud Platform

Code smell knowledge collected from different sources, is organized and made accessible on cloud platform. A new code smell web application is designed using Angular, Material Design, Node Js, Express JS and MongoDB for organization of code smell knowledge. Angular is an application design framework and development platform for

creating efficient and sophisticated mobile and desktop single-page applications. Some screenshots of code smell web application are given in Fig. 4.

The application is available on Heroku cloud platform at https://serene-tundra-28026.herokuapp.com and is under construction. All the tables from 1 to 11 with details are available in the site.

5 CONCLUSION

Code smell topic requires to be understood in depth. The objective of this exploratory study is to explore the code smell problem and its related concepts. A code smell repository is designed and code smell knowledge is arranged systematically. It is accessible on cloud platform. It enables developers and practitioners to set up a powerful foundation for exploring their idea about code smells. Also, this study can help other researchers to preserve a lot of time and resources. In the future, researcher plans to enhance the code smell repository by adding formulas and threshold values. Further this repository can be used to analyse the relationship between code smells and related concepts for identifying a minimal set of metrics, tools or refactoring actions to detect maximum set of code smells. Data mining techniques such as association rule mining can be used for finding representative software metrics for each code smell category. Clustering can be used to get a new way of categorizing code smells. Code smell repository and techniques of extracting insights from it can be made available to developers and practitioners.

6 REFERENCES

[1] Kaur, A. & Singh, S. (2018). Detecting Software Bad Smells from Software Design Patterns using Machine Learning Algorithms. *International Journal of Applied Engineering Research, 13*(11), 10005-10010.

[2] Singh, G. & Chopra, V. (2013). Design and implementation of testing tool for code smell rectification using c-mean algorithm. *International Journal of Advanced Research in Computer Science, 4*(9).

[3] Guggulothu, T. (2019). Code Smell Detection using Multilabel Classification Approach. *arXiv* preprint arXiv:1902.03222.

[4] Lanza, M. & Marinescu, R. (2007). *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media.

[5] Olbrich, S., Cruzes, D. S., Basili, V., & Zazworka, N. (2009). The evolution and impact of code smells: A case study of two open source systems. *The 3rd IEEE International Symposium on Empirical Software Engineering and Measurement*, 390-400. https://doi.org/10.1109/ESEM.2009.5314231

[6] Mannan, U. A., Ahmed, I., Almurshed, R. A. M., Dig, D., & Jensen, C. (2016). Understanding code smells in Android applications. *IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft 2016)*, 225-236. https://doi.org/10.1145/2897073.2897094

[7] Fowler, M. (2018). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

[8] Wangberg, R. (2010). A Literature Review on Code Smells and Refactoring. *Master's thesis*. Department of Informatics, University of Oslo, Norway.

[9] Humayoun, S. R., Hasan, S. M., Al Tarawneh, R., & Ebert, A. (2018). Visualizing software hierarchy and metrics over releases. *Proceedings of the 2018 International Conference on Advanced Visual Interfaces*, 1-5. https://doi.org/10.1145/3206505.3206548

[10] Srinivasan, K. P. (2015). Unique Fundamentals of Software Measurement and Software Metrics in Software Engineering. *International Journal of Computer Science & Information Technology (IJCSIT), 7*(4). https://doi.org/10.5121/ijcsit.2015.7403

[11] Eisty, N. U., Thiruvathukal, G. K., & Carver, J. C. (2018). A survey of software metric use in research software development. *The 14th IEEE International Conference on e-Science (e-Science 2018)*, 212-222. https://doi.org/10.1109/eScience.2018.00036

[12] Do Vale, G. A. & Figueiredo, E. M. L. (2015). A method to derive metric thresholds for software product lines. *The 29th IEEE Brazilian Symposium on Software Engineering*, 110-119. https://doi.org/10.1109/SBES.2015.9

[13] Fontana, F. A., Braione, P., & Zanoni, M. (2012). Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology, 11*(2), 5-1. https://doi.org/10.5381/jot.2012.11.2.a5

[14] Paiva, T., Damasceno, A., Figueiredo, E., & Sant'Anna, C. (2017). On the evaluation of code smells and detection tools. *Journal of Software Engineering Research and Development, 5*(1), 7. https://doi.org/10.1186/s40411-017-0041-1

[15] Fowler, M. (2003). *EtymologyOfRefactoring*. https://martinfowler.com/bliki/EtymologyOfRefactoring.html (accessed 16 October 2020).

[16] Webster, B. F. (1995). *Pitfalls of object-oriented development*. M and T books.

[17] Brown, W. J., Malveau, R. C., Brown, W. H., & McCormick, W. H. III, & Mowbray, T. J. (1998). *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley, 336 pages.

[18] Singh, S., & Kaur, S. (2018). A systematic literature review: Refactoring for disclosing code smells in object oriented software. *Ain Shams Engineering Journal, 9*(4), 2129-2151. https://doi.org/10.1016/j.asej.2017.03.002

[19] Fowler, M. & Beck, K. (2019). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 418 pages.

[20] Van Emden, E., & Moonen, L. (2002). Java quality assurance by detecting code smells. *Proceedings of the Ninth IEEE Working Conference on Reverse Engineering*, 97-106. https://doi.org/10.1109/WCRE.2002.1173068

[21] Kerievsky, J. (2005). *Refactoring to patterns*. Pearson Deutschland GmbH. https://doi.org/10.1007/978-3-540-27777-4_54

[22] Mäntylä, M. (2002). Experiences on applying refactoring. *Software Engineering Seminar*, 1-32.

[23] Mantyla, M. (2003). *Bad smells in software-a taxonomy and an empirical study*. Helsinki University of Technology.

[24] Li, W. & Shatnawi, R. (2007). An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *Journal of Systems and Software, 80*(7), 1120-1128. https://doi.org/10.1016/j.jss.2006.10.018

[25] Shatnawi, R. & Li, W. (2008). The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process. *Journal of Systems and Software, 81*(11), 1868-1882. https://doi.org/10.1016/j.jss.2007.12.794

[26] Fontana, F. A., Mariani, E., Mornioli, A., Sormani, R., & Tonello, A. (2011). An experience report on using code smells detection tools. *The Fourth IEEE International Conference on Software Testing, Verification and Validation Workshops*, 450-457. https://doi.org/10.1109/ICSTW.2011.12

[27] Ouni, A., Kessentini, M., Sahraoui, H., & Hamdi, M. S. (2012). Search-based refactoring: Towards semantics preservation. *The 28th IEEE International Conference on Software Maintenance (ICSM 2012)*, 347-356.

https://doi.org/10.1109/ICSM.2012.6405292
[28] Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., & De Lucia, A. (2014). Do They Really Smell Bad? A Study on Developers' Perception of Bad Code Smells. *Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME 2014)*, 101-110. https://doi.org/10.1109/ICSME.2014.32
[29] Pinto, G. H. & Kamei, F. 2013. What programmers say about refactoring tools? An empirical investigation of stack overflow. *Proceedings of the 2013 ACM Workshop on Refactoring Tools*, 33-36. https://doi.org/10.1145/2541348.2541357
[30] Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., & Poshyvanyk, D. (2015). When and why your code starts to smell bad. *The 37th IEEE/ACM International Conference on Software Engineering*, Vol. 1, 403-414. https://doi.org/10.1109/ICSE.2015.59
[31] Kaur, A. & Dhiman, G. (2019). A review on search-based tools and techniques to identify bad code smells in object-oriented systems. *Harmony search and nature inspired optimization algorithms*, Springer, Singapore, 909-921. https://doi.org/10.1007/978-981-13-0761-4_86
[32] Fontana, F. A., Lenarduzzi, V., Roveda, R., & Taibi, D. (2019). Are architectural smells independent from code smells? An empirical study. *Journal of Systems and Software*, *154*, 139-156. https://doi.org/10.1016/j.jss.2019.04.066
[33] Reis, J. P. D., Carneiro, G. D. F., & Anslow, C. (2020). Code smells detection and visualization: A systematic literature review. *arXiv* preprint arXiv:2012.08842
[34] Martins, J., Bezerra, C., Uchôa, A., & Garcia, A. (2020, October). Are code smell co-occurrences harmful to internal quality attributes? A mixed-method study. *Proceedings of the 34th Brazilian Symposium on Software Engineering*, 52-61. https://doi.org/10.1145/3422392.3422419
[35] Kaur, A. (2020). A systematic literature review on empirical analysis of the relationship between code smells and software quality attributes. *Archives of Computational Methods in Engineering*, *27*(4), 1267-1296. https://doi.org/10.1007/s11831-019-09348-6
[36] Al-Shaaby, A., Aljamaan, H., & Alshayeb, M. (2020). Bad Smell Detection Using Machine Learning Techniques: A Systematic Literature Review. *Arabian Journal for Science and Engineering*, *45*(4), 2341-2369. https://doi.org/10.1007/s13369-019-04311-w
[37] Alkharabsheh, K., Crespo, Y., Manso, E., & Taboada, J. A. (2019). Software Design Smell Detection: a systematic mapping study. *Software Quality Journal, 27*(3), 1069-1148. https://doi.org/10.1007/s11219-018-9424-8
[38] Lake, A. & Cook, C. (1994). Use of factor analysis to develop OOP software complexity metrics. *Proceedings of the 6th Annual Oregon Workshop on Software Metrics*, Silver Falls, Oregon.
[39] Vidal, S., Berra, I., Zulliani, S., Marcos, C., & Pace, J. A. D. (2018). Assessing the refactoring of brain methods. *ACM Transactions on Software Engineering and Methodology (TOSEM 2018), 27*(1), 1-43. https://doi.org/10.1145/3191314
[40] Fontana, F. A., Mangiacavalli, M., Pochiero, D., & Zanoni, M. (2015). On experimenting refactoring tools to remove code smells. *Scientific Workshop Proceedings of the XP2015*, 1-8. https://doi.org/10.1145/2764979.2764986
[41] Srinivasan, K. P. & Devi, T. (2014). A complete and comprehensive metrics suite for object-oriented design quality assessment. *International Journal of Software Engineering and Its Applications, 8*(2), 173-188.
[42] Alshayeb, M., Shaaban, Y., & Al-Ghamdi, J. (2018). SPMDL: software product metrics definition language. *Journal of Data and Information Quality (JDIQ), 9*(4), 1-30. https://doi.org/10.1145/3185049
[43] Shepperd, M. & Ince, D. (1993). *Derivation and validation of software metrics*. Oxford University Press, Inc.
[44] Sarker, M. (2005). *An overview of object oriented design metrics*. From Department of Computer Science, Umeå University, Sweden.
[45] Fenton, N. & Bieman, J. (2014). *Software metrics: a rigorous and practical approach*. CRC press. https://doi.org/10.1201/b17461
[46] Mori, A., Vale, G., Viggiato, M., Oliveira, J., Figueiredo, E., Cirilo, E., & Kastner, C. (2018). Evaluating domain-specific metric thresholds: an empirical study. *The IEEE/ACM International Conference on Technical Debt (TechDebt 2018)*, 41-50. https://doi.org/10.1145/3194164.3194173
[47] Núñez-Varela, A., Perez-Gonzalez, H. G., Cuevas-Tello, J. C., & Soubervielle-Montalvo, C. (2013). A methodology for obtaining universal software code metrics. *Procedia Technology, 7*, 336-343. https://doi.org/10.1016/j.protcy.2013.04.042
[48] Lanza, M. & Marinescu, R. (2007). *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media.
[49] Saranya, G. (2017). *Code smell detection and prioritization of refactoring operations to enhance software maintainability*. Faculty of Science and Humanities, Anna University.
[50] Refactoring.guru. (n.d.b). *Refactoring Techniques*. https://refactoring.guru/refactoring/techniques (accessed 16 October 2020)
[51] Slinger, S. (2005). *Code smell detection in Eclipse*. Delft University of Technology.
[52] Singh, M. & Salaria, D. S. (2013). Software defect prediction tool based on neural network. *International Journal of Computer Applications, 70*(22). https://doi.org/10.5120/12200-8368
[53] Virtual Machinery. (n.d.a). Object-Oriented Software Metrics - Class Level Metrics. Refactoring. http://www.virtualmachinery.com/jhawkmetricsclass.htm#:~:text=of%20Methods%20Called%20in%20class,Fan%20In%20and%20Fan%20Out (accessed 16 October 2020)
[54] Virtual Machinery. (n.d.b). Object-Oriented Software Metrics - Method Level Metrics. Refactoring. http://www.virtualmachinery.com/jhawkmetricsmethod.htm#:~:text=We%20can%20start%20at%20the,as%20a%20measure%20of%20productivity.&text=This%20is%20a%20measure%20of,through%20a%20piece%20of%20code (accessed 16 October 2020)

Authors' contacts:

Lida Bamizadeh, research scholar (Corresponding author)
Department of Computer Science, Savitribai Phule Pune University,
Ganeshkhind Rd, Ganeshkhind, Pune, Maharashtra 411007, India
9503039485, lida_bamizadeh@yahoo.com

Binod Kumar
JSPM's Rajarshi Shahu College of Engineering (MCA Dept.),
Tathawade, Pimpri-Chinchwad, Maharashtra 411033, India
9665548971, binod.istar.1970@gmail.com

Ajay Kumar
JSPM Jayawant, Technical Campus,
Tathawade, Pimpri-Chinchwad, Maharashtra 411033, India
7972095030, ajay19_61@rediffmail.com

Shailaja Shirwaikar
Department of Computer Science, Savitribai Phule Pune University,
Ganeshkhind Rd, Ganeshkhind, Pune, Maharashtra 411007, India
7066046154, scshirwaikar@gmail.com