International Journal on Future Revolution in Computer Science & Communication Engineering ISSN: 2454-4248
Volume: 4 Issue: 3 62 – 74

Identifying Various Code-Smells and Refactoring Opportunities in Object-Oriented Software System: A systematic Literature Review

Randeep Singh
Department of Computer Science &Engineering,
Maharishi Markandeshwar University, Mullana- Ambala,
133-207 (Haryana),India
randeeppoonia@gmail.com

Dr. Ashok Kumar
Department of Computer Science & Engineering
Maharishi Markandeshwar University, Mullana- Ambala,
133-207 (Haryana),India
mailtodr.ashok@redikkmail.com

***Abstract*—** Software maintenance is a prolonged and necessary phase in software development in order to incorporate the changing functional and non-functional requirements of the user. It causes the architecture of a software system to drift away from its original design and results in increased maintenance effort and cost. Poor design architecture is caused by bad programming practice adopted by the programmers and bad design adopted by them. The segments of the source code that causes the degraded software architecture is known as code smells. The code smells must be identified and mitigated in order to improve the underlying software architecture and hence overall quality of a software system. One approach commonly adopted is known as Refactoring of the associated code segment in the source code of a software system. Different refactoring opportunities aim at locating the locations of different code smells in source code and removing them in order to finally improve the underlying software architecture, hence the quality of the software system. Code smells identification and applying corresponding refactoring opportunities being the hot research area in software engineering, so, it must be systematically surveyed in order to summarize the extensive literature material already present. Therefore, this research paper aims at presenting a systematic literature review of existing approaches and techniques related to code smell detection and various associated refactoring approaches. The literature survey performed is extensive in nature and covers high-quality relevant research papers since 2010. The conclusions presented in this paper are very beneficial for the researchers engaged in this field and different industry personals engaged in software development.

*****

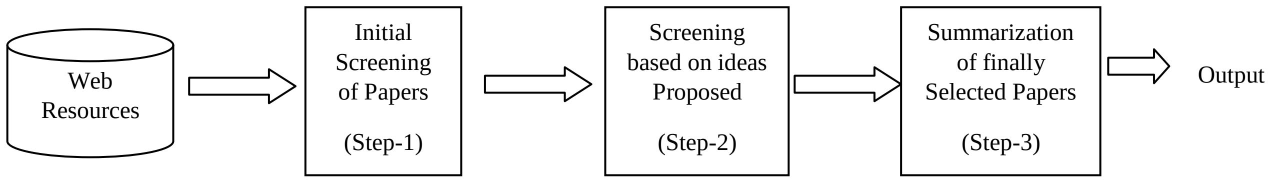
INTRODUCTION

Software quality is the key research area in software engineering and various maintenance activities during software development has asignificant negative impact on the overall quality of software system. Quick and careless modification performed by maintenance team results in thedeterioration of the code quality and as a result, our code starts to smell. This smell causes the maintenance overheads to be increased significantly. In literature, the causes of the bad smell in thecode are termed as Code-Smells and the associated techniques to overcome them are termed as Refactoring.This paper aims at presenting a systematic extensive survey in this field.

The rest of the paper is organized as follows: section 2 describes the methodology adopted by us in order to perform systematic literature survey. Section 3 and 4 details about code-smells and refactoring techniques. Finally, section 5 concludes the paper.

METHODOLOGY

This research paper aims at providing a systematic literature survey about the code smells presented in thesource code of a software system and the refactoring techniques that can be utilized to overcome those code smells. The methodology adopted in this paper is shown in figure-1 and consist of mainly three steps that are systematically followed:


Web Resources
Initial Screening of Papers (Step-1)
Screening based on ideas Proposed (Step-2)
Summarization of finally Selected Papers (Step-3)
Output

Initial Screening of Papers: this is the first step of our methodology and it starts with downloading of high-quality research articles from online libraries of high repute such as ACM, ScienceDirect, WILEY, IEEE XPLORE, SPRINGER etc. For this purpose, we constituted a team consisting of 6 students belonging to graduate, postgraduate degrees. The students were expert at their level and initially, the team has selected and downloaded a total of 255 research articles. The paper downloaded at this stage are one which contains key keywords such as code smell, refactoring, code quality, bad smell.

Screening based on Ideas Proposed: in next phase of our systematic study, we performed further scrutiny of the selected research articles based on the idea proposed by them on the basis of data provided in abstract and conclusion sections. The careful analysis done in this stage results in 105 papers left for further study.

Summarization of finally selected papers: Finally 105 papers are thoroughly studied and analyzed. Based on the thorough study, we collected information about what is abad smell, types of bad smell, toolsand technique used to detect and mitigate a particularly bad smell. Finally, all types of bad smell and various refactoring techniques are classified into different groups and results are presented.

CODE-SMELLS

It is very important to have an idea about the **code smell** what it actually is. Code smell is known as thebad smell but in computer programming codes the code smell resembles a symptom in thecode of any program which indicates adeep problem. “Code smell is a surface indication corresponds to a deeper problem in any system “the line said by Martin flower. Code smells have some structures that show the violation of principle design and thenegative impact of design quality”. The concept of Code Smell is given by Martin flower and beck as an indicator of problems in the design or code of software like bad code indicates the factor to atechnical problem in any code. List of code smell is termed as “value system “for software. The code does not help to correct any problem and in the process of prevention it’s only work is to indicate the problems in any design which slowdowns the development and also indicates the risk area from bugs. The code should not just be sufficient cable of producing the appropriate result but also it should be written in such a format that, it should take minimum effort to produce the appropriate result. Code smell generally indicates that the code should be refactored or the overall design should be re-examined. AS our topic is code smell and refactoring now we discussabout the refactoring. Refactoring is a process changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence when you re-factor you are improving the design of the code after it has been written.*”* Refactoring has become a well-known technique for the software engineering community. Martin Fowler has defined it as a process to improve the internal structure of a program without altering its external behavior. Frequent refactoring of the code helps theprogrammer to make the code more understandable, find bugs and make it suitable for the addition of new features and to program faster. Above all that, it improves the design of the software and therefore the overall quality of the software. Refactoring can be done manually as well as automatically.It is incorrect to compare Code Smells with bugs—the part containing code smell is not syntactically/ semantically incorrect and it does not hinder the program normal functionality. Instead, they pinpoint to design flaws which may slowdown thefuture expansion of the software can be the cause of future bugs, system failures.

Code Smell	**Description**	**Refactoring**
Alternative Classes with Different Interfaces	When another class doing the same job is created just for a different signature. This should be replaced with overloading and mergeinto a single class.	Rename Method, Move Method
Comments	While comments are great, refactoring may make most comments superfluous. If a method is heavily commented, it may need refactoring	Extract Method, Introduce Assertion
Data Class	A class that contains data, but hardly any logic for it is Data class	Move Method, Encapsulate Collection, Encapsulate Field
Data Clumps	Data clumps the smell in which the software that has the data items often appear together.	Extract Class, Preserve theWholeObject, Introduce Parameter Object
Divergent Change	The one class which needs to be continuously changed for the different reasons is called the Divergent Change.	Extract Class
Duplicated code	Identical or very similar code exists in more than one location, bad because if you modify one instance of duplicated code but not the others, you (may) have introduced a bug!	Extract Method, Extract Class, Pull UP Method, Form Template Method
Feature envy	A class that uses methods of another class excessively.	Move Method, Move Field, Extract Method

Inappropriate Intimacy	Sometimes classes become far too cozy (friendly) and put in a lot of time inquiring each other.	Move Method, Move Field, Association to Unidirectional, Hide Delegate, Change Bidirectional, Replace Inheritance with Delegation
Incomplete Library Class	When library functionality doesn't provide the complete set required (usually when you are given a library to work with external to your organization or project).	Introduce Foreign Method, Introduce Local Extension
Large class	a class that has grown too large. Classes try to model/ provide too much functionality resulting in ultimately decreased cohesion score of the class.	Extract Class, Replace Data Value with Object, Extract Subclass, Extract Interface
Lazy Class	A class is called the Lazy class that is doing nothing enough and should be removed.	Inline Class, Collapse Hierarchy
Long Method	A long or complex method is amethod which is too long harder to read and modify which is difficult to extend, change and understand.	Extract Method, Replace Temp with Query
Long Parameter List	A long list of parameters as anargument in a function/ method make readability and code quality worse.	Replace Parameter with Method, Preserve theWhole Object, Introduce Parameter Object
Message Chains	When there is a long sequence of internal method calls on behalf of a client requesting services of a particular object.	Hide Delegate
Middle Man	Middleman,it means that a class is delegating most of its tasks to the subsequent classes	Remove Middle Man, Replace Delegation with Inheritance, Inline Method
Parallel Inheritance Hierarchies	Whenever a subclass of a class is created then automatically there is a need to make asubclass of another class also. Parallel inheritance hierarchies are the special case of shotgun surgery.	Move Method, Move Field
Primitive Obsession	Where the small classes represent the smell by the case of primitives.	Replace Data Value With Object, Replace Type Code With Subclass, Replace Type Code With Class
Refused Bequest	In this, the methods or data it inherits are not fully supported by thechild class.	Replace Inheritance with delegation
Shotgun Surgery	Shotgun surgery is alike to divergent change except it is opposed.	Move Method, Move Field, Inline class
Speculative Generality	The use of theoretical classes or super methods which must be overridden to be useful.	Collapse Hierarchy, Rename Method, Inline Class, Remove Parameter
Switch Statements	Too many switch statements show procedural statements.	Replace Conditional with Polymorphism, Replace Parameter with Explicit Methods, Replace Type Code with Subclasses, Introduce Null Objects
Temporary Field	The smell means that class has a variable which is only used in some situation.	Introduce Null Object, Extract Class

REFACTORING

Refactoring is a technique which is well known for the Software engineering. Martin Fowler has defined it as a process to enhance the underlying architecture of the software without modifying its external functionality/ working.

Refactoring Techniques:

1. Composing Methods

2. Moving Features between Objects
3. Organizing Data
4. Simplifying Conditional Expressions
5. Making Method Calls Simpler
6. Dealing with Generalization
7. Big Refactoring

(1) **Composing Methods**: The large part of the refactoring is composing methods to code and almost the problems come from methods that are too long.

(a) *Extract Method:* To take out a code chunk into a separate group called themethod. Create a new method and name it by what it does not by how it does.

(b) *Inline Method:* Replace the method call statement with the method's body and delete the actual method defined.

(c) *Inline Temp:* A temporary variable which is evaluated only once and having a simple expression then that temporary variable use is replaced with the corresponding expression.

(d) *Substitute Algorithm:* Consist of replacing the code statements performing a given functionality (coded in the form of algorithmic steps) with a clearer/ simpler new algorithm implementing the same functionality.

(e) *Replace Temp with Query:* It is similar to extract method in these Replace short-term variables with the method calls. Replace all the references to the temp with expressions.

(f) *Introduce Explaining Variable:* Decomposing a complicated expression into parts and storing the result in a temporary variable which is having a name that correctly defines the purpose in order to improve the readability.

(g) *Split Temporary Variable:* If a temporary variable is assigned in different expressions at multiple places, then make a separate temporary variable for each of such kind of assignments.

(h) *Remove Assignments to Parameters:* Remove Assignments to Parameters means the code allocated to the parameter and used the temporary variable in its place.

(i) *Replace Method with Method Object*: If a long method has a large number of local variables which hinders in applying Extract Method refactoring then we replace the method with a new separate class having the said member variables as members. This helps in splitting the original method into anumber of sub-methods.

(2) **Moving Features between Objects:** In this refactoring technique explain towards securely go functionality among classes, create new classes, and also hide execution facts from public access.

(a) *Hide Delegate:* If a client is directly interacting with the delegate instead of an interface on server, then modify the client to call methods redefined on server on behalf of delegate.

(b) *Remove Middle Man*: In this method a class is performing a direct delegation without any kind of modification then replace the middle man class and directly call the delegate method straight way.

(c) *Introduce Foreign Method*: In this method a server class that are using requirements of an added method, except adjust the class. In this create a class which contains these extra methods as server class needs extra methods but we are not able to change the class.

(d) *Introduce Local Extension:* In Introduce Local Extension, the server class that needs more than a few supplementary methods, other than modify the class.

(e) *Move method*: The Move method means moving a method to an extra class when classes have too much functionality to do as method is using more features of other class rather than the class in which it is defined.

(f) *Move field:* Move the used field of a class into another class which uses this field directly. A field is used by another class than the class on which it is defined.

(g) *Extract Class:* A one class doing work that should be done by two.

(h) *Inline Class:* If a give class isn't performing much appropriate functionality. In this case shift all its fields and methods to some other class along with deleting it.

(3) **Organizing Data:** The Organizing Data techniques facilitate with data handling and replace primitives with loaded class functionality.

(a) *Self-Encapsulate Field:* Self-Encapsulate Field in this field directly, accessing but the combination to the field is attractive self-conscious i.e. coupling to the field is becoming awkward.

(b) *Replace Data Value with Objects:* If a particular data has its own associated fields then convert the data from field to a new class along with the object creation i.e. covert the data item into object if it needs additional behavior.

(c) *Change Value to Reference:* Change Value to Reference has a class with many identical instances that desire to put back with the single object (Reference object).

(d) *Change Reference to Value:* In this a position of object that is minute, absolute, and uncomfortable to manage then turn it into the value object.

(e) *Replace Array with Object*: In this if a given array field is storing multiple types of data in it then we create a new object for the said array and definite a new field for each type of data in the given array.

(f) *Duplicate Observed Data:* In this if a GUI based class is storing both GUI information and associated data then it is beneficial to separate both so as to ensure synchronization.

(g) *Change Unidirectional Association to Bidirectional:* In this two classes wants to use each other's functionality, but if there exist only a one-way connection then it is useful to add missing link with a back pointer, and also alter the modifiers to update both sets.

(h) *Change Bidirectional Association to Unidirectional:* In this there exists both-way connection between two classes and now one of the class don't needs to use features of other then delete the extra link of the connection between two classes.

(i) *Replace Magic Number with Symbolic Constant:* In this factual number with a strict meaning are replaced with a suitable constant name or a method call which ultimately provides the static number back.

(j) *Encapsulate Field:* If a class contains a public field then convert it to private and provide assessors.

(k) *Encapsulate Collection:* Encapsulate Collection is a method which precedes a collection and creates it to give back (get method) a read-only sight & provide an add element (set method) methods so that it can only return read-only view rather than reference collection.

(l) *Replace Record with Data Class:* In this the boundary through a record structure inside a usual programming surroundings and make dumb information for the record i.e. make a dumb data object for the record.

(m) *Replace Type Code with Class:* A class that has a numeric type code which does not affect the behavior of the class. Replace the number with a new class.

(n) *Replace Type Code with Subclasses:* In this the absolute type code which involve the data or behavior of a set then replace the type code with subclasses as that code may affect the behavior of the class.

(o) *Replace Type Code with State/Strategy:* In this the type codes which involve the performance of a class and cannot use sub classing, replace it with state object.

(p) *Replace Subclass with Fields:* If a given subclass only differs in methods that return only constant data that go back steady. Delete the subclasses.

(4) **Simplifying Conditional Expressions:** Conditionals are inclined to obtain more and more complex in their reason over time. Since a large amount of the conditional activities is handling by means of polymorphism.

(a) *Decompose Conditional:* Different effects depending on various conditions, so fast finish up with an attractive long method. Used Extract methods for the complicated conditional statement.

(b) *Consolidate Conditional Expression:* If there exists a sequence of conditional tests that are producing the same results then combine all of such conditional statements into a single expression that does the same job.

(c) *Replace Conditional with Polymorphism:* An object its type chooses different behavior depending on qualified conditional and Move every support of conditional to a prime method within a subclass. It can be implemented by making the original method abstract and used theoverriding method in a subclass.

(d) *Consolidate Duplicate Conditional Fragments:* In this if multiple branches of a conditional statement have same set of statements than move the common set outside to improve readability i.e. move it outside of the expression to remove the same fragment of the code.

(e) *Remove Control Flag:* In this if a variable is performing the job of control flag for a set of Boolean statements then replace that flag with a much easier and clearer break/ return statement instead.

(f) *Replace Nested Conditional with Guard Clauses:* A method has restricted activities which do not construct the clear normal pathway of execution and used protector section for all the unusual cases. If there is an unusual condition, then check the condition and return back if the condition become true otherwise not

(g) *Introduce Null Object:* Introduce Null Object which has frequent checks for null price i.e. repeated checks for null values then substitute or replace the null value with a null object

(h) *Introduce Assertion:* In this a segment of code assumes about the situation of the program and makes the statement explicit with an assertion. It is assumed that the assertion i.e. the conditional statement is always true otherwise it will indicate the error in the program.

(5) **Making Method Calls Simpler:** It simplifies the interfaces designed between classes.

(a) *Rename method:* Provides a new convenient name for the method if the name doesn't define its purpose.

(b) *Add parameter:* If a given method requires additional information from its visitor then add this information as a new parameter used for the same purpose so as to exceed under this information.

(c) *Remove parameter:* Remove any unnecessary parameter from method body which is no longer required in its body.

(d) *Separate Query from Modifier:* Separate a method into two parts if it simultaneously queries as well as modifies any field in an object i.e. one method for query and another method for the modification.

(e) *Parameterize Method:* A number of methods do similar things but with dissimilar values controlled in the method body. Here, creates only one method that uses argument for the different values.

(f) *Replace Parameter with Explicit Methods:* In this a method is divided into different methods based on the parameter used i.e. concerned set of statement based on a given parameter are moved to another method.

(g) *Preserve Whole Object:* In this the set of values are passed from an object rather them passing as individual to method's parameter list i.e. send the whole object.

(h) *Replace Parameter with Method:* In this an object call upon a method, then get ahead of the effect as a parameter, intended for a method i.e. passes the result as a argument for the method.

(i) *Introduce Parameter Object:* In this a repeated list of parameters are replaced by a single object containing all the required fields.

(j) *Remove Setting Method:* This is a field which set information only at the time of creation and it does not undergone modification afterwards then remove the concerned method which set the new value for that field.

(k) *Hide Method*: If a method's functionality defined within a class is not used outside of its class then hide its visibility by making it as a private method within a class.

(l) *Replace Constructor with Factory Method:* This method has a simple structure when you create an object and then replace the constructor with a factory method i.e. replace type code with subclasses.

(m) *Encapsulate Downcast:* In this, method precedes an object that requirements to be downcasted by its callers i.e. move the downcast to within the method.

(n) *Replace Error code with Exception:* A method that precedes an individual code to point to an error and fling an exception instead.

(o) *Replace Exception with Test:* In this throwing a checked exception on a condition the caller could have checked first, change the caller to make the test first.

(6) **Dealing with Generalization:** Generalization produces a set of refactoring, typically trade by means of affecting methods about a ladder of inheritance.

(a) *Pull up Field:* In this move the common fields of two subclasses to its parent superclass.

(b) *Pull up Method:* This technique moves the common methods of two subclasses into its parent class.

(c) *Pull up Constructor Body:* This method contains constructors on subclasses through theypically equal body so creates a superclassconstructor call this preliminary the subclass methods.

(d) *Pull down Method:* Pull down Method is the method which is reverse of Pull up Method.

(e) *Push Down Field:* If a field is used only by a few of its subclasses rather than all of its subclasses then move the concerned member field into its associated subclasses along with deletion from its parent superclass.

(f) *Extract Subclass:* Move the features of a class that are necessary only in few of its child classes to a new subclass. This prevents unnecessary inheritance of features from its parent class.

(g) *Extract Superclass:* If there are two classes in asystem with have comparable features then make one class as superclass and move the features which are common in the superclass.

(h) *Extract Interface:* Extract Interface is applied if more than two classes have the same subset class interface, or two classes contain a common division of their boundary in terms of features.

(i) *Collapse Hierarchy:* If a superclass and corresponding child class have no major difference then merges them together by collapsing the parent-child relationship.

(j) *Form Template Method:* In this the two techniques in descendent class that perform comparable ladder in the same order and the steps are dissimilar, put the steps into methods which is having the same signature so that the main methods become the same.

(k) *Replace Inheritance with Delegation:* If the descendent class overrides only a few services defined by superclasses interface or we do not want to use inheritance then we simply create the new concerned field/ method of the superclass into child class afresh. This avoids unnecessary inheritance.

(l) *Replace Delegation with Inheritance:* Replace Delegation with Inheritance which regularly marks a lot of easy delegations meant for the whole interface.

(7) **Big Refactoring:** All the previously discussed refactoring techniques mainly focuses on the minute size refactoring applied, but there are refactoring techniques that are applied at big scale e.g. extract hierarchy.

(a) *Tease apart Inheritance:* In this, the inheritance hierarchy which is liability doing two jobs at one time and creates two hierarchies for used designation to appeal to one class from the other class.

(b) *Extract Hierarchy:* In this a class with the intention of doing too much work through lots of conditional statements.

(c) *Convert Procedural Design to Objects:* In this code written in procedural style. Turn the data records into objects, break up the behavior, and move the behavior to the objects.

(d) *Separate Domain from Presentation:* In this the GUI classes that enclose domain logic then separate the field logic into divide into domain classes.

Refactoring techniques:

1) Techniques permitted for abstraction

- **Encapsulate field-** It forces the code to proceed by using getter and setter methods.
- **Generalize type-** It creates a number of general types which helps in code sharing.
- Substitute type checking code with strategy
- Substitute conditional with polymorphism.

2) Techniques used for breaking codes into logical pieces

- **Componentization:** It breaks the code into thesemantic unit in areusable manner.
- **Extract class:** Helps to move code part from old (existing) class to new class.
- **Extract method:** It converts thelarge method into new method by breaking codes into small, pieces.

3) Techniques used to improve the names and location of code

- **Move method:** Moves code to appropriate class.
- **Rename method:** It changes the old name andgives new name according to their purpose.
- **Pull-up**: It moves the code to superclass it is an object-oriented programming.
- **Push-Down**-Use to move to sub-class.

Refactoring Opportunities

The most frequently applied refactoring identification approaches are quality metrics-oriented, precondition-oriented, and clustering-oriented. Code slicing has been shown to be an effective approach in several software engineering areas, such as code testing and quality measurement. We have identified six different approaches followed by the SPs to empirically evaluate the identification techniques-

1. Intuition-Based Evaluation
2. Quality-Based Evaluation
3. Mutation-Based Evaluation
4. Comparison-Based Evaluation
5. Behavior-Based Evaluation
6. Applicability-Based Evaluation

LITERATURE REVIEW

There is alarge quantity of research material which is already available in theliterature. Different researchers have proposed different techniques to detect code smell and refactor them. Some of the researchers have also proposed new types of code smells that is present in various modern programming language and can affect the overall quality significantly. Also, the refactoring solution to different code smells is tackled using various techniques. The summary of the various techniques already proposed by different researchers is as follows:

Mantyla, et al [1] represented a taxonomy that categorized common bad smell. In this paper, they assume that taxonomy makes the smell understandable and recognizes the relationship between smells. In a small Finnish software product company, an empirical study is done for evaluating code quality. The result shows that taxonomy can explain the correlation between the subjective evaluations of the existence of the smell. Peters, et al [2] proposed a solution that is based on the use of anti-patterns to remove code smells that impacted code quality. In this paper, they assume that code smells are the symptoms of anti-pattern which occurs at thecode level. In this paper, they use to do a case study where they investigate the lifespan of code smell and refactoring behaviors in 7 open systems. This investigation helps to solve the code smell and anti-patterns. The result shows that engineers are aware of code smells but not too much with their impact. Sjoberg, et al [3] performs aninvestigation on the relation between code smell and maintenance effort. In this paper, they hired six developers which perform 3 maintenance tasks each on 4 java system implemented by different companies. Every developer devotes its 3 to 4 week in the given task. They modified 300 Java files of 4 systems. After adjustment for the size of thefile, the result shows that not even a single smell out of 12 investigations associated with increased effort whereas refused bequest was associated with thedecreased effort. Hermans et al. [4] use a known code smell in aspreadsheet formula. Here they present a list of matrices with help of which they can determine smelly formula and highlight these formulas in spreadsheet they use visualization techniques. To evaluate our approach in 2 different ways they implement the metrics and given technique in a prototype tool. To study the occurrence of the formula smell they analyze EUSES spreadsheet corpus and to identify smell they use to analyze and interview of 10 real life spreadsheet. The result shows that formula smells are common and can reveal real errors and weaknesses in spreadsheet formulas. Kataoka, et al. [5] proposed an approach to measure the maintainability enhancement effect of program refactoring a quantitative evaluation method is given. To evaluate refactoring effect the whole concentration is on coupling the metrics. They can evaluate the degree of maintainability enhancement by comparing the coupling before and after the refactoring. The result shows that this method was really effective to quantify the refactoring effect and helped us to choose appropriate refactoring.

Mantyla et al. [6] give an empirical study on subjective evaluation of bad code smell. The concluded result shows theidentify poor structure in software. Two contributions are done after the empirical study 1) there is a study of evaluator effect while evaluating existence smell in thecode module. The study shows that it is quite difficult to determine code by using smells 2) to identify three smells code metrics is applied and compare results to the subjective evaluations. Du Bois, et al. [7] performs an analysis of how refactoring manipulates coupling/cohesion characteristics. There is an analysis on identifying refactoring opportunities which improve these characteristics. In this paper, they introduce a practical guideline for the optimal usage of refactoring in a software maintenance process. Munro, et al [8] uses different software metrics regarding the identification of the characteristics of bad smell is concentrated. Here to interpret the software metrics result which is going to apply on Java source code they use the predefined set of interpretation rule. Here firstly from the informal description given by the originator an accurate definition of bad smell is given. To enable the evaluation of the interpretation rules a prototype tool is used. Mealy, et al. [9] performs an analysis of the task of software by using ISO 9241-11. By expanding the analysis, they obtain 11 collections of theguideline and then they combine these into a single list of 38 guidelines. They use to develop 81 requirements for refactoring tools from the given list. They can study Eclipse, condenser, Simian by using these tools. They select a subset of the requirement which can be incorporated into prototype refactoring tool intended to address the full refactoring process on the basis of this analysis. Khomh, et al. [10] performs an investigation on code smell that is if code smell is more change prone than the classes without thesmell. They use to do a test on the above question. Here they determine 30 codes smell in 9 releases of azures, 13 in eclipse and there is a study on the relation between classes with these codes smells and class change-proneness. The result shows that releases of Azureus and Eclipse, classes with code smells are more change-prone than others. Palomba, et al. [11] use exploiting change history information mined from versioning system to detect five different code smell i.e. divergent change, shotgun surgery, parallel inheritance, blob and feature envy. This HIST is applied to 8 software projects which are written in Java and then compared with existing state-of-art on the basis of code analyses. The result shows that Accuracy of HIST ranges from 61- 80 %. The outcome confirms that HIST can identify code smell. Fontana, et al. [12] proposes various tools to detect code smell every individual is characterized by particular features. The main goal of this paper is to give a description of the experience to detect code smell. Here they highlight the main difference between them and obtain adifferent result. Tsantalis, et al. [13] in an exhibition, displayed an Overshadowing module that consequently distinguishes Sort Checking awful stenches in Java source code, and resolves them by applying the Supplant Contingent with Polymorphism¿ or ¿ Supplant Sort Code with State/Strategy refactoring. To the best of our insight, there is an absence of devices that recognize Sort Checking awful stenches. Also, none of the best in class IDEs bolster the refactoring that determination such sort of awful stenches.

Fokaefs, et al [14] introduce an Overshadowing module that distinguishes include begrudge awful stenches in Java tasks and resolves them by applying the proper move technique refactoring. The fundamental commitment is the capacity to pre-assess the effect of all conceivable move refactoring's on plan quality and apply the best one. Kadar et al [15] give a DataSet to be used to analyze refactoring. Based on the data set, they analyze the relationship between maintainability and refactoring activities. Analyzed how refactoring affects various metrics computed from source code at the method level. They conclude that performing refactoring helps in removing unwanted code smells and gives more maintainable architecture of the software system. Arendt, et al. [16] presented a tool that models the quality aspect of the software system based on the Eclipse Modeling Framework (EMF). The framework proposed enforces software quality. They use various metrics to model various code smells and then uses different refactoring steps.

Santiago, et al. [17] detected that most of the existing tools detect a wide variety of code smells in software and they are difficult to handle together. To solve this problem, they present a semi-automated technique that lets developers concentrate on the critical code smells of asoftware system. They ranked various types of code smells present in the software system based on three factors namely: past history of modification applied to the component, important regarding current modification needs of the software, and importance of the type of code smell. Fontana, et al. [18] studied the ability of various machine learning algorithms in detecting the various types of code smells through a large experimentation carried out. They experimented and studied only four types of code smells related to source code (Feature Envy, Data Class, Large Class, and Long Method). Mkaouer, et al. [19] models the Refactoring Problem as multi-objective and uses NSGA-II to solve it. Performed refactoring over a software system and during this process they have chosen an optimal balance between quality and robustness parameters of the software system. Considered the uncertainties regarding the correction applied to various code smells during the software development process (dynamic environment). In thedynamic environment of software development, it is not reasonable to consider the severity of code smell and associated class importance as fixed. Almugrin, et al. [20] Uses Indirect Coupling measure at thepackage level. Proposes measurement of Ripple Effect. Gives flow based metrics to measure maintainability at thepackage level. The scope of Structural, lexical dependencies analysis.

Al-Shara, et al. [21] proposed an approach for recovering underlying architecture from OO source-code. Uses the recovered architecture to model component's descriptions. Also gives amethod to describe interfaces of the modeled components. Divides classes into two sets: Internal Classes & Boundary Classes. Gives method to transform generated components to well-known component-based language (OSGi/ SOFA). Corazza, et al. [22] uses lexical information from source-code to perform clustering and to finally recover architecture. Extracted lexemes form six zones. Uses probabilistic-based approach (Expectation-Maximization Algorithm) for weighing lexemes. Perform clustering using theK-Medoid algorithm. Hasheminejad, et al. [23] proposed a clustering-approach named CCIC (Clustering analysis Classes to identify software Components) for identifying various logical components by considering the underlying architectural preferences of thesystem. The proposed approaches make use of tailored HEA (Hierarchical Evolutionary Algorithm) to automatically cluster the underlying classes into appropriate logical components. Zhao, et al. [24] proposed a new package-modularization metrics. They further used them to calculate software fault-proneness. Compared proposed new package-modularization metrics with traditional package-level metrics. Bavota, et al. [25] proposed a technique for are-modularizing software system. Utilizes latent topics in source code and the structural dependencies together with Relational Topic Model (RTM). Provides only move class refactoring opportunities by moving the said class to a more appropriate package. Provides opportunities for the involvement of software developers in there-modularization process. The scope of studying thesame concept at lower levels i.e. classes. Bavota, et al. [26] proposed an automatic approach for re-modularization of a software system's package by distributing it into a new/ subpackage. The author's considered the structural and semantic characteristics of the system in order to divide a package into more cohesive ones. Consider only individual package for decomposition. Tufano, et al. [27] studied various reasons which are responsible for the presence of code smells in software systems. The authors also studied the location of adifferent kind of code smells in a given source code. For this purpose, they have studied over 200 opens source software systems by considering their different versions into account also.

Santos, et al. [28] reveals that in order to identify the God Class code smell in a software system an empirical study is mandatory. The authors further stressed that an active participation of human being can highly improve the detection of God Class code smell in a software system. Kaur Sharanpreet, et al. [29] carried out a vast study on real-world software called JHotDraw for a smell called "type checking code smell" using JDeodorant software tool which is present as the Eclipse plugin. The input software chosen for study JHotDraw is a large sized project currently used as theindustry standard. Chen Jie, et al. [30] proposed an Iterative development model for refactoring the software system as per the requirements are given by the user. The author further surveyed in order to determine various refactoring practices that must be adopted by programmers in future. GhannemAdnane, et al. [31] proposed a genetic algorithm based methodology for detecting and mitigating various kinds of code smells from the Object-Oriented software system. They have applied the proposed approach on four standard open-source software systems viz Gantt Project, Xerces-J, QuickandJHotDraw. Fontana Francesca, et al. [32] proposed a new machine learning based methodology for rectifying the bad smells present in a given software system. The authors studied a large set of open-source software data along with more than 15 set of different machine learning approaches in order to detect and rectify the code smells. The authors stressed that machine learning approaches are really helpful in code smell area (with anaccuracy of >90%). Gatrell M, et al. [33] studied the impact of refactoring with faults present in the software system and the corresponding changes performed in thesystem after refactoring. The bespoke tool was used for C# projects and detected total 15 types of code smells. ZhaoYangyang, et al. [34] proposed a new package-level metric that is helpful in identifying the fault in a given software system. The package level metric is developed from traditional metrics. Palomba Fabio, et al. [35] proposed a new approach called HIST for identifying five types of code smells namely– Divergent Change, Shotgun Surgery, Parallel Inheritance, Blob, and Feature Envy in a given software system. SaraivaJulianade AG, et al. [36] performed an in-depth survey regarding maintainability in OO Software systems by studying different metrics at thecode level. They have categorized the metric into different 17 groups based on the expert's review of this field and their impact on maintainability. Lozano Angela, et al. [37] studied the inter-relations that exists among adifferent kind of code smells and how they are coupled. They have studied relation among four categories of code smells.

Han Ah-Rim, et al. [38] proposed an approach to identify and remove the move method refactoring in software to improve its maintenance. Their approach was based on the Maximal independent set (MIS) theory to identify the refactoring. The open source projects studied includes Edit, Columba, and Jgit. Gaitani Maria Anna G, et al. [39] proposed an automatic approach to identify and rectify the null object design pattern present in a software system. The proposed automatic approach is based on a newly proposed algorithm. This algorithm is made publically available in the form of Eclipse Plug-in. Ammar Boulbaba Ben, et al. [40] proposed a new refactoring approach to identify and correct the problematic areas present in an OO software system. The proposed approach makes use of UML, B,and CSP. Class diagrams are used by authors in order to validate the software structure. Palomba Fabio, et al. [41] proposed a new approach to detect and rectify long method code smell based on the textual analysis of conceptual data present in a software system. The new approach is called TACO (Textual Analysis for Code smell detection). Ujhelyi Zoltan, et al. [42] performed a vast evaluation (25 projects) for estimating the performance of a software system based on the anti-pattern detection techniques. Anti-patterns are detected using incremental local search based refactoring technique. Morales Rodrigo, et al. [43] proposed a new framework to automatically correct the anti-patterns in a software system. The proposed framework makes use of thesearch-basedrefactoring technique to rectify the anti-patterns. Liu Hui, et al. [44] proposed an approach to apply Rename Refactoring opportunity in an OO software system. The proposed approach is based on semantic analysis of the source code and is available as an eclipse plugin tool known as “Rename Expander”. Misbhauddin Mohammed, et al. [45] surveyed 3295 research articles related to UML based refactoring models. They reveal that UML based refactoring techniques have a significant research scope in near future. Morales Rodrigo, et al. [46] surveyed regarding the factors that affect thequality of a software system. They concluded that 7 kinds of anti-patterns are responsible for thepoor design of a software system and thus directly effect on the quality.

JaafarFehmi, et al. [47] studied regarding therelationship between faults and anti-patterns in a software system based on identified design patterns. They concluded that from versions to versions of a software systems, anti-patterns adapt themselves and keeps on existing in the software system. Fenske Wolfram, et al.[48] based on the experimental study proposed new variability-aware code smells that exist in an OO software system. They also concluded that code smells are directly related to understandability and maintainability of a software system. Abílio Ramon, et al. [49] identified God Method, Shotgun Surgery and God Class code smell that are present in an AHEAD programming language (a powerful FOP language). Palomba Fabio, et al. [50] revealed based on astudy that the literature lacks in supporting tools for the detection and correction of code smells. The authors further proposed a new online platform tool called LANDFILL which is capable of detecting bad smells in a system. Ounia Aliet al. [51] proposed a set of refactoring practices that are capable of preventing smells in a system. The proposed guidelines are for four types of smells. The studied systems include – Gantt Project, JHotDraw, ArtOfIllusion, JFreeChart,and Xerces-J. Kaur Sharanpreet, et al.[52] conducted an extensive literature survey regarding techniques/ approaches to detect various kinds of code smells and antipatterns. They found that there are basically four major directions regarding the approaches that have been already followed by many researchers regarding detection of bad smells. NetoBaldoino, et al.[53] proposed a new tool called ‘‘Auto- Refactoring” which can automatically find out code smells and apply the necessary refactoring actions. The studied software systems that are open source include viz Log4j, Sweet-Home3D, HSQLDB, jEdit8,and Xerces. Zhao Song, et al. [54] surveyed a new domain called parallel refactoring and provide its technicalities. They have concluded various challenges that a maintenance team faces while converting a sequential program to corresponding parallel code. Yamashita Aiko, et al. [55] performed a vast survey regarding theimpact of bad smells on the software maintenance. They have studied it under expert guidelines regarding maintenance. Liu Hui, et al. [56] studied the impact of software code smells on its maintenance cost and concluded that around 17.64–20% efforts can be minimized if the associated code smells are identified timely by the developer team. The authors further suggest that inexperience in development generally give rise to potential flaws in thecode of a software system. Murphy-Hill Emerson, et al. [57] proposed a new refactoring tool, that based on the suggestions taken as input, apply them on the OO code and also capable of raising an error message occurring during this process. Chu Peng-Hua, et al. [58] proposed a new test case based refactoring approach for XP. The proposed approach is based on patterns applied and the validation technique used for the study. Fokaefs Marios, et al. [59] proposed a new automatic extract method refactoring technique which is based on theidentification of the slice and object state slice identification techniques. The proposed approach is provided publically as an Eclipse plug-in. The extract method refactoring is based on the identification of the Feature Envy code smell. Singh Satwinder, et al. [60] proposed a metric model that based on measurement of different metrics at source code levels helps in identifying classes having a possible presence for code smells in the system. The obtained results indicate that source code metrics are helpful in detecting code smells with higher accuracy in different classes belonging to a software system. ChristopoulouAikaterini, et al. [61] proposed an automatic approach which applies refactoring to mitigate various bad smells in asoftware system. The proposed approach of refactoring makes use of an important Eclipse plug-in called JDeodorant for detecting the code smells present.

CONCLUSION

This survey paper presents systematic and concise details about code smells and various refactoring techniques by doing through study. In this paper, the factors affecting the overall quality of an Object-Oriented software system is studied. Various kind of code-smells that are responsible for deteriorating overall quality of underlying software system are extensively studied. Based on the study, a thorough literature is presented describing: what is code-smells, how it deteriorate the software quality, types of code-smells, techniques to overcome the bad smell present in the underlying software system. One of the technique to tackle bad smell in asoftware system is known as arefactoring of the underlying sourcecode. So, this paper also provides details about refactoring and its associated terminologies in detail. Refactoring detection at non-source code level can be done in software design model, source code with non-conventional detection, and other software artifacts. The methods used can be categorized as aheuristic method, where the refactoring identification is done based on certain rules, and non-heuristic method, where the refactoring identification is done with acertain algorithm that explores every possibility of refactoring opportunities. The studies which propose neither method are generally surveyed papers about refactoring catalog or comparative review.

REFERENCES

[1]. Mantyla, Mika, JariVanhanen, Casper Lassenius, "A taxonomy and an initial empirical study of bad smells in code", International Conference on.IEEE, 2003.

[2]. Peters, Ralph,AndyZaidman, "Evaluating the lifespan of code smells using software repository mining", Software Maintenance and Reengineering (CSMR), 16th European Conference on. IEEE, 2012.

[3]. Sjøberg, Dag IK, "Quantifying the effect of code smells on maintenance effort", IEEE Transactions on Software Engineering, vol. 39, no. 8, pp. 1144-1156, 2013.

[4]. Hermans, Felienne, Martin Pinzger, Arie van Deursen, "Detecting code smells in spreadsheet formulas",Software Maintenance (ICSM), 28th IEEE International Conference on.IEEE, 2012.

[5]. Kataoka, Yoshio, "A quantitative evaluation of maintainability enhancement by refactoring", Software Maintenance, Proceedings.International Conference on.IEEE, 2002.

[6]. Mantyla, Mika V., JariVanhanen, Casper Lassenius, "Bad smells-humans as code critics", Software Maintenance, 2004.Proceedings.20th IEEE International Conference on.IEEE, 2004.

[7]. Du Bois, Bart, Serge Demeyer, and Jan Verelst."Refactoring-improving coupling and cohesion of existing code." Reverse Engineering, 2004.Proceedings.11th Working Conference on.IEEE, 2004.

[8]. Munro, Matthew James, "Product metrics for automatic identification of" bad smell" design problems in java source-code." Software Metrics, 11th IEEE International Symposium.IEEE, 2005.

[9]. Mealy, Erica, et al.,"Improving usability of software refactoring tools" ,Software Engineering Conference ,ASWEC ,18th Australian. IEEE, 2007.

[10]. Khomh, Foutse, Massimiliano Di Penta, Yann-Gael Gueheneuc, "An exploratory study of the impact of code smells on software change-proneness" ,Reverse Engineering, WCRE'09,16th Working Conference on.IEEE, 2009.

[11]. Palomba, Fabio, et al.,"Detecting bad smells in source code using change history information." Automated software engineering (ASE), IEEE/ACM 28th international conference on.IEEE, 2013.

[12]. Fontana, Francesca Arcelli, et al., "An experience report on using code smells detection tools" Software Testing, Verification and Validation Workshops (ICSTW), IEEE Fourth International Conference on IEEE, 2011.

[13]. Tsantalis, Nikolaos, Theodoros Chaikalis, Alexander Chatzigeorgiou, "JDeodorant: Identification and removal of type-checking bad smells" Software Maintenance and Reengineering, CSMR 12th European Conference on.IEEE, 2008.

[14]. Fokaefs, Marios, NikolaosTsantalis, Alexander Chatzigeorgiou, "Jdeodorant: Identification and removal of feature envy bad smells" Software Maintenance IEEE International Conference on IEEE, 2007.

[15]. I. Kadar, P. Hegedus, R. Ferenc, T. Gyimothy, “Assessment of the Code Refactoring Dataset Regarding the Maintainability of Methods”, ICCSA 2016, Part IV, LNCS 9789, Springer, pp. 610–624, 2016.

[16]. T. Arendt, G. Taentzer, “A tool environment for quality assurance based on the Eclipse Modeling Framework”, Autom Softw Eng, vol. 20, pp. 141–184, 2013.

[17]. S. A. Vidal, C. Marcos, J. A. Diaz-Pace, “An approach to prioritize code smells for refactoring”, Autom Softw Eng, vol. 23, pp. 501–532, 2016.

[18]. F. A. Fontana, M. V. Mäntyla, M. Zanoni. A. Marino, “Comparing and experimenting machine learning techniques for code smell detection”, Empir Software Eng, vol. 21, pp. 1143–1191, 2016.

[19]. M. W. Mkaouer, M. Kessentini, S. Bechikh, M. Cinnéide, K. Deb, “On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach”, Empir Software Eng, vol. 21, pp. 2503–2545, 2016.

[20]. Almugrin S, Albattah W, Melton A,”Using indirect coupling metrics to predict package maintainability and testability”, Journal of Systems & Software, Elsevier, 2016.

[21]. Al-Shara Z, Seriai A.D, TibermacineC,”Materializing Architecture Recovered from OO Source Code in Component-Based Languages”, 10th European Conference on Software Architecture (HAL), 2016.

[22]. Corazza A, Martino S, Maggio V, Scanniello G,” Weighing lexical information for software clustering in the context of

architecture recovery", Empirical Software Engineering, 2016.

[23]. Hasheminejad S.M.H, Jalili S," CCIC: Clustering analysis classes to identify software components", Information & Software Technology, Elsevier,2015.

[24]. Zhao Y, Yang Y, Lu H, Song Q, "An empirical analysis of package-modularization metrics: Implications for software fault-proneness", Information & Software Technology, Elsevier,2015.

[25]. Bavota G, Gethers M, OlivetoR,"Improving Software Modularization via Automated Analysis of Latent Topics & Dependencies" ACM Trans. Softw. Eng,2014.

[26]. Bavota G, Lucia A.D, Marcus A,"Using structural and semantic measures to improve software modularization", Empirical Software Engineering,2013.

[27]. Tufano Michele, Palomba Fabio, Bavota Gabriele, Oliveto Rocco, Penta Massimiliano Di, Lucia Andrea De, Poshyvanyk Denys, "When and why your code starts to smell bad", IEEE/ACM 37th International Conference on Software Engineering, 2015.

[28]. Santos, MMendonça Jose Amancio, CleberManoel, Gomes De Pereira,"The problem of conceptualization in god class detection: agreement, strategies and decision drivers", J Softw Eng, 2014.

[29]. Kaur Sharanpreet, Singh Satwinder," Spotting & eliminating type checking code smells using eclipse plug-in: JDeodorant", Int J Comput SciCommunEng, vol. 5, no.1, 2016.

[30]. Chen Jie, Xiao Junchao, Wang Qing, Osterweil Leon J, Li Mingshu," Perspectives on refactoring planning and practice: an empirical study", Empir Softw Eng, vol. 21, no. 3, 2016.

[31]. GhannemAdnane, BoussaidiGhizlane El, KessentiniMarouane," On the use of design defect examples to detect model refactoring opportunities", Softw Qual J, Springer, 2015.

[32]. Fontana Francesca Arcelli, Mantyla Mika V, Zanoni Marco, Marino Alessandro, " Comparing and experimenting machine learning techniques for code smell detection", Empirical Softw Eng, 2015.

[33]. Gatrell M, Counsell S," The effect of refactoring on change and fault-proneness in commercial C software", Science of computer programming, vol. 102, pp. 44-56, 2015.

[34]. Zhao Yangyang, Yang Yibiao, Hongmin Lu, Zhou Yuming, Song Qinbao, XuaBaowen," An empirical analysis of package-modularization metrics: Implications for software fault-proneness", Inf Softw Technol, vol. 57, pp.186-203, 2015.

[35]. Palomba Fabio, Bavota Gabriele, Di Penta Massimiliano, Oliveto Rocco, Poshyvanyk Denys," Mining version histories for detecting code smells", IEEE Trans Softw Eng, vol. 41,no. 5,pp. 462-89, 2015.

[36]. SaraivaJulianade AG, DeFrançaMicael S, Soares Sergio CB, Filho Fernando JCL, DeSouza Renata MCR," Classifying metrics for assessing object-oriented software maintainability: a family of metrics catalogs", J Syst Softw, vol. 103,pp. 85-101, 2015.

[37]. Lozano Angela, Mens Kim, Portugal Jawira, "Analyzing code evolution to uncover relations between bad smells", IEEE, pp. 2-5, 2015.

[38]. Han Ah-Rim, Bae Doo-Hwan, Cha Sungdeok," An efficient approach to identify multiple and independent Move Method refactoring candidates", Inf Softw Technol, vol. 59, pp. 53-66, 2015.

[39]. Gaitani Maria Anna G, ZafeirisVassilis E, Diamantidis NA, Giakoumakis EA, "Automated refactoring to the NULL OBJECT design pattern", Inf Softw Technol, vol. 59 ,pp. 33-52, 2015.

[40]. Ammar Boulbaba Ben, Bhiri Mohamed Tahar," Pattern-based model refactoring for the introduction association relationship", J King Saud University – Comput Inform Sci, vol. 27, no. 2, pp. 170-80, 2015.

[41]. Palomba Fabio," Textual analysis for code smell detection", IEEE/ACM 37th IEEE international conference on software engineering, pp. 769–71, 2015.

[42]. Ujhelyi Zoltan, Szoke Gabor, Horvath Akos, Csiszar Norbert Istvan, Vidacs Laszlo, Varra Daniel, Ferenc Rudolf, "Performance comparison of query-based techniques for anti-pattern detection", Inf Softw Technol, vol. 65, pp. 147-65, 2015.

[43]. Morales Rodrigo," Towards a framework for automatic correction of antipatterns", IEEE, pp. 603-04, 2015.

[44]. Liu Hui, Liu Qiurong, Liu Yang, Wang Zhouding," Identifying renaming opportunities by expanding conducted rename refactoring", IEEE Trans Softw Eng, pp.55-89, 2015.

[45]. Misbhauddin Mohammed, Alshayeb Mohammad," UML model refactoring: a systematic literature review", Empirical Softw Eng, vol. 20, pp. 206–51, 2015.

[46]. Morales Rodrigo, McIntosh Shane, KhomhFoutse, "Do code review practices impact design quality? A case study of the Qt", VTK and ITK Projects IEEE, pp. 171–80, 2015.

[47]. JaafarFehmi, Gueneuc Yann-Gael, Hamel Sylvie, KhomhFoutse, Zulkernine Mohammad, "Evaluating the impact of design pattern and anti-pattern dependencies on changes and faults", Empirical Softw Eng, 2015.

[48]. Fenske Wolfram, Schulze Sandro," Code smells revisited: a variability perspective", Proceedings of the ninth international workshop on variability modeling of software-intensive systems, Germany Copyright ACM, pp. 3:3–3:10, 2015.

[49]. Abílio Ramon, Padilha Juliana, Figueiredo Eduardo, Costa Heitor," Detecting code smells in software product lines – an exploratory study", 12th international conference on information technology – new generations, pp. 433–8, 2015.

[50]. Palomba Fabio, Nucci Dario Di, Tufano Michele, Bavota Gabriele, Oliveto Rocco, Poshyvanyk Denys, "Lucia Andrea De. Landfill: an open dataset of code smells with public evaluation", 12th working conference on mining software repositories, IEEE, 2015.

[51]. Ounia Ali, KessentiniMarouane, Sahraoui Houari, Inoue Katsuro, SalahHamdi Mohamed," Improving multi-objective code-smells correction using development history", J Syst Softw, vol. 105, pp. 18– 39, 2015.

[52]. Kaur Sharanpreet, Singh Satwinder,” Influence of anti-patterns on software maintenance: a review”, International conference on advancements in engineering and technology (ICAET), pp. 14–19, 2015.

[53]. NetoBaldoino Fonseca dos Santos, Ribeiro Marcio, Silva Viviane Torres da, Braga Christiano, Lucena Carlos Jose Pereira de, Costa Evandro de Barros,” AutoRefactoring: a platform to build refactoring agents”, Expert SystAppl, vol. 42, no. 3, pp. 1652–64, 2015.

[54]. Zhao Song, BianYixin, Zhang Sen ,” A review on refactoring sequential program to parallel code in Multicore Era”, International Conference on Intelligent Computing and Internet of Things (IC1T), 2015.

[55]. Yamashita Aiko, Moonen Leon, “Do code smells reflect important maintainability aspects”, IEEE IntConf Softw Maint, 2012.

[56]. Liu Hui, Ma Zhiyi, Shao Weizhong, NiuZhendong,” Schedule of bad smell detection and resolution: a new way to save effort”, IEEE Trans Softw Eng, vol. 38, no. 1, 2012.

[57]. Murphy-Hill Emerson, Black Andrew P,” Programmer-friendly refactoring errors”, IEEE Trans Softw Eng, vol. 38, no. 6, pp. 1417–31, 2012.

[58]. Chu Peng-Hua, Hsueh Nien-Lin, Chen Hong-Hsiang, Liu Chien-Hung,” A test case refactoring approach for pattern-based software development”,SoftwQual J, vol. 20, no. 1, pp. 43-75, 2012.

[59]. Fokaefs Marios, Tsantalis Nikolaos, StrouliaEleni, Chatzigeorgiou Alexander,” Identification and application of Extract Class Refactorings in object-oriented systems”, J Syst Softw, vol. 85, no. 10, pp. 2241–60, 2012.

[60]. Singh Satwinder, Kahlon KS,” Effectiveness of encapsulation and object-oriented metrics to refactor code and identity error prone classes using bad smells”, ACM Sigsoft Soft Eng. Notes, vol. 37, no. 2, pp. 1-10, 2012.

[61]. ChristopoulouAikaterini, Giakoumakis EA, ZafeirisVassilis E, SoukaraVasiliki,” Automated refactoring to the Strategy design pattern”, Inform Soft Technol, vol. 54, pp. 1202–14, 2012.