Ain Shams Engineering Journal 9 (2018) 2129–2151

Contents lists available at ScienceDirect

Ain Shams Engineering Journal

journal homepage: www.sciencedirect.com

Engineering Physics and Mathematics

A systematic literature review: Refactoring for disclosing code smells in object oriented software

Satwinder Singh [a], Sharanpreet Kaur [b,*]

[a] Central University of Punjab, Bathinda, India
[b] IKGPTU, Jalandhar, India

ARTICLE INFO

Article history:
Received 3 June 2016
Revised 6 December 2016
Accepted 13 January 2017
Available online 22 March 2017

Keywords:
Code smells
Anti-patterns
Refactoring

ABSTRACT

Context: Reusing a design pattern is not always in the favor of developers. Thus, the code starts smelling. The presence of "Code Smells" leads to more difficulties for the developers. This racket of code smells is sometimes called Anti-Patterns.
Objective: The paper aimed at a systematic literature review of refactoring with respect to code smells. However the review of refactoring is done in general and the identification of code smells and anti-patterns is performed in depth.
Method: A systematic literature survey has been performed on 238 research items that includes articles from leading Conferences, Workshops and premier journals, theses of researchers and book chapters.
Results: Several data sets and tools for performing refactoring have been revealed under the specified research questions.
Conclusion: The work done in the paper is an addition to prior systematic literature surveys. With the study of paper the attentiveness of readers about code smells and anti-patterns will be enhanced.

1. Introduction & motivation

For achieving excellent quality a regular maintenance and smart development is needed. Various quality assurance activities could be practical like review, code walks, testing and refactoring. These could be applied when the software is created for the first time or when the already existing resources are used like source code or design templates. According to Fowler [1], "Refactoring is a process of altering the internal layout of software without making the changes in the external behavior". The process could be applied by software engineers by using the patterns of code time and again. Except, some of the cases the patterns launches up in the appearance of "ANTI-PATTERNS". These are the defects which are called code smells or anti-patterns. There is a very diminutive difference between the anti-pattern and code smell. However, code smells are technically not erroneous but their presence point towards flimsiness in design, which could initiate the malfunction of system and risk of bugs in the near future. Fowler [1] defined different types of code smells which requires Refactoring. Code Smells is a kind of warning for the presence of anti-patterns. Hence anti-patterns are assumed to be a pessimistic clarification that will be generating more trouble than they solve. In this paper a systematic literature review is prepared which will focus on the different kinds of techniques for the identification of code smells. Such kinds of reviews are generally time overwhelming but helps in endowing with crystal clear and broad view of enduring research. It could also impart different opportunities to researchers in the field of code smells detection, the basic concepts and techniques.

1.1. Motivation

(a) Detection of smells will improve the quality assurance activities which are needed for maintenance in the software.
(b) To enable software developers with the insight of code smells, as it is among one of the least informed issue in the software industry.
(c) While going through the literature survey it has been detected that there is a lack of systematic review with large dataset in the field of code smells detection. Therefore we explore the on hand research using the existing database, to reduce the research gaps.

* Corresponding author.
E-mail addresses: satwindercse@gmail.com (S. Singh), sharancgm@gmail.com (S. Kaur).
Peer review under responsibility of Ain Shams University.

Production and hosting by Elsevier

http://dx.doi.org/10.1016/j.asej.2017.03.002
2090-4479/

The leftovers of the paper are arranged as follows: The paper is divided into eight sections. Section 2 discloses the background of work, general preface to code smells and the categorization of code smells and disadvantages of code smells. Section 3 describes the research planning for the systematic literature survey which specifies the research questions, sources of information and the inclusion exclusion criteria. Section 4 explores the code smell detection approaches. Section 5 concludes the results according to the research questions suggested by the study. Section 6 provides the insinuation of work. Section 7 concludes the paper with future recommendations.

2. A background of related work

Various authors have studied the impact of refactoring on the source code and detection of code smells and anti-patterns. However, only a few have performed the systematic literature survey in the field of code smell detection and anti-patterns. Few milestones are provided by the authors like Al Dallal [2], Misbhauddin and Alshayeb [3], Zhang et al. [4] and Mens and Toure [5] in the discipline of refactoring and code smell detection.

Misbhauddin and Alshayeb [3] revealed the investigation of literature using more than 30 papers. Inclusion-Exclusion Criteria had been applied at four levels. Different perspectives were explored like impact of refactoring on code quality and tools used for refactoring of UML models. Different search engines were chosen like Google Scholar, Wiley Inter Science Journal Finder, Science Direct, Elsevier etc. for revealing several code smell detection approaches. However the author targeted the design oriented (UML diagrams) refactoring possibilities.

Zhang et al. [4] performed a vast literature survey to study bad smells in code. More than 300 papers were investigated from the year 2000 to 2009. Different research parameters were investigated and presented from different perspectives. Leading journals from IEEE, ACM, Springer and others were searched. The author revealed that Duplicated Code Smell is considered by most of the researchers than any other smell. Mens and Tourwe [5] acted upon a survey on refactoring and its implications on software quality. Various tools for refactoring were revealed with the type dataset of source code used in the articles. The author performed the refactoring of a code example throughout the paper using the refactoring possibilities suggested by Fowler [1]. Various approaches have been discussed with the advancements of tools. The proposed study is different from the study of Zhang et al. as it concentrates on different parameters like refactoring, code smells, anti-patterns, object oriented design and refactoring and software maintenance with respect to code smell detection etc. Moreover Min Zhang performed the survey five years ago. Similarly, in Mens and Tourwe paper [5] the refactoring activity is primarily focused on extensive tools.

Al Dallal [2] performed the systematic literature survey on the possibilities of performing refactoring in Object Oriented Systems. The primary focus is on the detection of code smells. The investigation of more than 45 studies had been done since 2013. Various approaches for the detection of smells were brought into limelight. The work revealed the open source systems potentially used by the researchers. The author concluded that from the set of all the open source projects JHotDraw was used by most of the researchers to validate their results. Java language was found as the most used language by developers for performing refactoring.

Apart from the above mentioned systematic literature surveys, the degree of work made in the last few years has been remarkably boosted. Therefore the proposed work is an extension of the work done earlier by the researchers. However, the size of the dataset for this systematic literature survey is comparatively larger in size than the dataset used by Al Dallal [2]. Hence, there is a great need for a systematic literature survey in the field of code smell detection.

The systematic review in the paper has followed the strategy of Kitchenham et al. [6], Brereton et al. [7], and Stapic et al. [8]. According to the above mentioned researchers, the systematic literature survey is a way of exploring the available literature according to the set of well defined questions and the concerned subject. But before going into the details of work we need to summarize refactoring, code smells and anti-patterns, categories of code smells. We then sum up with the future recommendations and suggestions for the research community in the field of code smell detection.

2.1. Refactoring

At the outset we will define Refactoring, Needs of Refactoring, Code smells (indication of refactoring) and their types. The term refactoring was firstly revealed by Opdyke in 1992 [9]. Fowler [1] wrote the book along with Opdyke on Refactoring. In this book they disclosed Refactoring as an activity which picks up the internal layout of software but the external behavior still remain the same. According to Martin Refactoring could be used either as a noun or as a verb. A good refactoring makes the software easier to use & understand.

Refactoring defined as ***Noun***: It is defined as an alteration in the inner organization of the software by making the external layout the same.

Refactoring defined as ***Verb***: Having the set of Refactoring Methods applied on the software without affecting the external actions.

2.2. Code smells

Firstly the term anti-pattern was instigated by Opdyke [9]. Code smells are the indicator of the presence of anti-patterns in the code. There is a very undersized gap between anti-pattern and code smell. Both the terms anti-patterns and code smells are used interchangeably. Anti-patterns are supposed to be a dreadful programming practice but not an error. A short experience and extraneous knowledge of the software developers leads to the involvement of code smells into the literature, for solving an exact problem. Code Smell is an allusion that admits anti-patterns. Though code smells are technically not wrong but they indicate delicacy in design. The presence of which will lead to the bugs in the near future.

Brown et al. [11] and Webster [12] revealed code smells like Blob, Spaghetti Code, Conditional Complexity, Anti-Singleton, Class Data Should Be Private (CDSBP), Refused Parent Bequest (RPB), Swiss Army Knife, etc. Fowler specified 22 code smells are mentioned in Table 1 along with possible solutions.

2.3. Categorization of code smells

Mantyla [13] categorizes code smells into broader units such that each individual unit reflects a familiar impression. All the above mentioned 22 code smells are divided into seven units, which are depicted clearly in Table 2.

(a) **The Bloaters** – Classes that became enormously larger in size and difficult to manage. Such type of classes is difficult to manage and cannot be effectively handled. However, the size could increase at both the class and method level.
(b) **The Object-Orientation Abusers** – Classes that are not justifying with the rules of object oriented programming like maximal use of Switch Statements in the code lead to violation of few OOP concepts.

Table 1
Code smells and refactoring possibilities.

S No.	Name of smell	Solution
1.	Duplicated Code	*Extract Method, Pull Up Method, Substitute Algorithm.*
2.	Long Method	*Extract Method, Replace Temp With Query, Introduce Parameter Object and Preserve Whole Object. Replace Method with Method Objects, Decompositional Objects*
3.	Large Class	*Extract Class, Extract Sub Class. Duplicate Observed Data*
4.	Long Parameter List	*Replace Parameter with Method*
5.	Divergent Change	*Extract Class*
6.	ShotGun Surgery	*Move Class or Move Method*
7.	Feature Envy	*Move Method and Extract Method*
8.	Data Clumps	*Extract Class, Introduce Parameter Object and Preserve Whole Object*
9.	Primitive Obsession	*Replace Data Value With Object, Replace Type Code With Class, Replace Type Code With Subclass., Replace Type Code With State Strategy, Extract Class, Introduce Parameter Object And Replace Array With Object.*
10.	Switch Statements	*Extract Method, Move Method, Replace Type Code with Subclass or Replace Type Code with State*
11.	Parallel Inheritance Hierarchies	*Move Method or Move Field*
12.	Lazy Class	*Collapse Hierarchy or Inline Class*
13.	Speculative Generality	*Collapse Hierarchy, Inline Class, Remove Parameters Methods, Remove Methods*
14.	Temporary Fields	*Extract Class, Introduce Null Objects*
15.	Message Chains	*Hide DELEGATE, Extract Method and Move Method.*
16.	Middle Man	*Remove Middle Man, Inline Method, Replace Delegation with Inheritance*
17.	Inappropriate Intimacy	Move Method or Move Field, Change Bidirectional Association to Unidirectional, Extract Class, Hide Class
18.	Alternative Classes with different interfaces	*Rename Method, Move Method, Extract Superclass*
19.	Incomplete Library Class	*Move Method, Introduce Foreign Method, Introduce Local Extension*
20.	Data Class	*Encapsulate Field or Encapsulate Collection, Remove Setting Method, Move Method or Extract Method, Hide Method*
21.	Refused Bequest	*Push Down Method Push Down Field, Replace Inheritance with Delegation*
22.	Comments	*Extract Method or Rename Method, Introduce Assertion.*

Table 2
Units of code smells.

S No.	Unit	Code smells involved
1	Bloaters	Long Method, Large Class, Data Clumps, Primitive Obsession and Long Parameter List
2	Object Orientation Abusers	Switch Statements, Temporary Field, Refused Bequest and Alternative Classes with Different Interfaces
3	Change Preventers	Divergent Change, Shotgun Surgery and Parallel Inheritance Hierarchies
4	Dispensables	Lazy class, Data class, Duplicate Code, Dead Code and Speculative Generality
5	Encapsulators	Message Chains and Middle Man
6	Couplers	Feature Envy and Inappropriate Intimacy
7	Others	Incomplete Library Class and Comments

(c) **The Change Preventers** – Classes in which further changes in the software product lead to enormous errors like Divergent Change and Shotgun Surgery. However both the above mentioned code smells behaved differently from each other.
(d) **The Dispensables** – Classes which contain Superfluous elements which are not demanded by an efficient source code.
(e) **The Encapsulators** – The smells in this category increased at the cost of each other i.e. if both the smells are present in the code, then decreasing one smell lead to automatically increasing other smell.
(f) **The Couplers** – Classes that reveal elevated coupling hence violate the principle of Object Oriented Programming. Feature Envy and Inappropriate Intimacy code smell reveal high coupling in the code hence disobeying the concept of OOP. Thus, these smells should be avoided first.
(g) **Others** – Smells which do not suit to all of the above mentioned units are placed under this category. Good examples are smells related to comments, for instance.

2.4. Disadvantages of code smells

The presence of bad smells in the code introduces drawbacks that make the code error prone and difficult to manage. Some of the flaws due to presence of code smells are mentioned by few authors.

(a) van Emden and Moonen [14] proposed the negative influence of code smells on the software product. They suggested a methodology which could reduce the impact of code smells on java source code. The work resulted in code smell browser called "Jcosmo".
(b) Olbrich et al. [15] evaluated the performance degradation of open source projects in the presence of smells. This negative temperament of bad smells was tested for Log4j, Apache Lucene and Apache Xerces. "EvolutionAnalyzer" was provided for the detection of smells.
(c) van Emden and Moonen [16] found that the quality of software was affected at most by the existence of smells.
(d) Tufano et al. [17] explored the reason for bad smells in the code. A survey of hundreds of projects was done to unmask the problem of bad smells.
(e) Ganea et al. [20] revealed the negative impact of bad smells in the code. The work introduced an Eclipse plug-in called "InCode" which helps in declining the bad code smells and leads to high quality Java software. Code smells like Code Duplication, God Class, Feature Envy and Data Class could be easily located by InCode.
(f) Yamashita and Moonen [21] assessed that bad smells impact negatively the software maintenance. For the validation of the work, software was placed under the observation of the experts for maintenance.
(g) Fontana et al. [22] revealed the affect of code smells on the quality of software. The authors have given names of a few smells that are present in most of the software systems – Duplicate Code, Data Class, God Class, Schizophrenic Class and Long Method.

(h) Liu et al. [60] suggested a strategy to avoid the influence of code smells software. The authors claimed up to 17.64–20% efforts can be reduced if the presence of bad smells was detected in time.

3. Research method

The review protocol consists of construction of research questions, defining the set of databases which are to be searched, collection of data, and analysis of data and discussions of findings. The aim of the systematic literature survey is to identify the gaps done by researchers earlier to this survey.

3.1. Methodology of the review

The review protocol comprises of designing the research questions, finalizing the databases which are to be searched and analysis of discussions and findings. The process includes the search of primary and supplementary databases, applying the inclusion-exclusion criteria, generating the results and concluding with the discussions. To keep the protocol development fair, the work done by one author is examined carefully by the second one. In case of conflict the opinions are openly discussed and resolved by iteration and findings. For the literature survey both the electronic and manual databases are searched including the foremost journals, conference proceedings and thesis of researchers. Overall 325 articles were acknowledged during the search process; upon applying the inclusion-exclusion criteria only 238 are selected as shown in Fig. 1 (see Fig. 3).

3.2. Research questions

The main aim of systematic literature review is to disclose the existing literature spot lighting refactoring, code smells, object oriented design and refactoring, code smell or anti-pattern detection and different techniques used for code smell detection. A set of research questions has been proposed to conduct systematic review mentioned in Table 3.

3.3. Sources of information

In order to implement a systematic review a wider outlook is needed. So before starting the review a proper set of databases are to be selected, which could easily provide the appropriate results according to the relevant keywords. We selected the following four databases.

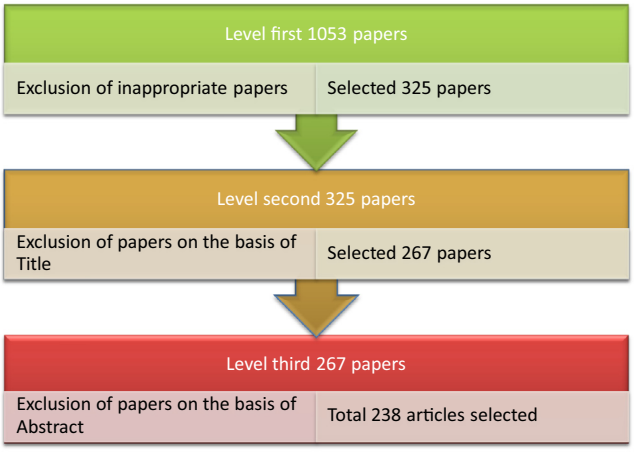

Figure 1. Search exclusion criteria.

(a) IEEE eXplore (http://ieeexplore.ieee.org/).
(b) Springer (http://www.springer.com/in/).
(c) ACM Digital Library (http://dl.acm.org).
(d) ScienceDirect (http://www.sciencedirect.com/).

3.3.1. Supplementary sources

(a) Conference Proceedings.
(b) Books Chapters.
(c) Published Technical Reports.
(d) Thesis of researchers.
(e) Review Articles.

3.4. Vital keywords for research

We searched all the primary and supplementary databases for the specific set of keywords. The search was performed between August to September 2015. The keywords given below that have been used for each source.

(a) Refactoring
(b) Code smell
(c) Anti-pattern
(d) Object Oriented Design AND Refactoring
(e) Fault Tolerance AND Object Oriented Design
(f) Software Metrics AND Anti-patterns
(g) Anti-patterns in Cross Company Projects
(h) Software Maintenance AND Code Smells

3.5. Inclusion and exclusion criteria

The inclusion exclusion-criteria have been applied on three levels where the unrelated papers were discarded from the search. We included only the papers related to computer science and engineering. However, as the word "patterns" is multidisciplinary in nature so while searching papers from different branches were also found.

All those other articles are of different subjects like medical, food processing, material sciences, biomechanics and nanotechnology. Such types of papers were discarded. We included only the papers written in English. The systematic review incorporated research papers from the initial dates of the digital libraries to September 2015. However same research papers from different libraries are discarded so as to make the search stable. The serial research papers published from the same authors with some modifications are considered. Similarly if a paper is presented in a Conference Proceedings and later published in a premier journal, is also considered. The systematic review consists of three levels as given in Fig. 1. We found 1053 papers in the first round, which we refined to 325 papers based on the title of the paper. Later on, from the set of these papers we selected 267 papers based on their abstract. Finally 238 papers are extracted based on the full text of the paper. Finally a set of 238 papers were preferred after applying inclusion and exclusion criteria.

3.6. Quality evaluation of research

The quality of the systematic literature survey is examined after the inclusion exclusion criteria have been finalized. The quality is evaluated according to the guidelines of Kitchenham [6] and Brereton [7]. Each module is validated for the parameters defined by the author like bias, internal validity and external validity. The papers selected are assessed for the quality evaluation after the application of Inclusion-Exclusion Criteria. Three appendices – Appendices A–C are incorporated for achieving a quality oriented systematic literature survey. Basic information about the system-

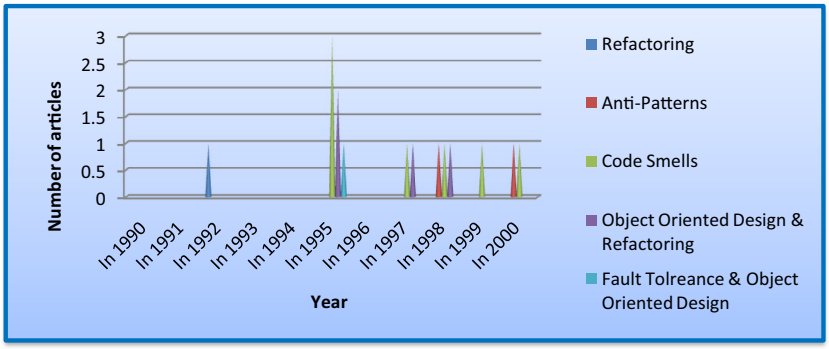

Figure 2. Year wise description of papers (1990–2000).

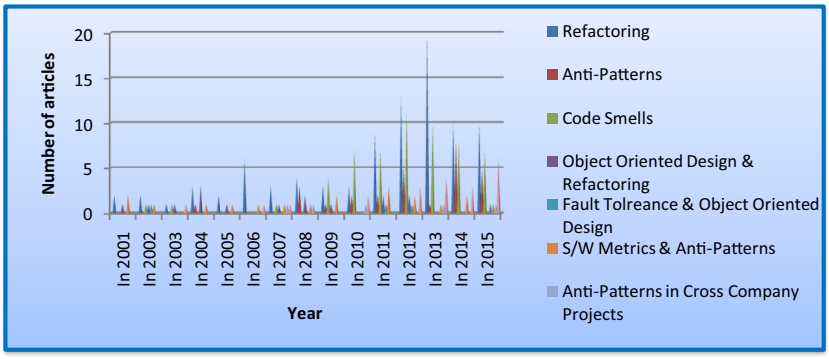

Figure 3. Year wise description of papers (2001–2015).

Table 3
Research questions.

RQ1. What is the current status of refactoring with respect to code smells and anti-patterns?
RQ2. What are the different approaches used for the detection of code smells and how the smells are removed using these approaches?
RQ3. What are the different tools used by the researchers to identify code smells?
RQ4. What are the different datasets used by the authors in order to detect code smells?
RQ5. What are the different types of code smells spotted in the papers?

atic survey is validated in Appendix A. If the reviewer is satisfied with the first quality form i.e. Appendix A, then he proceeded to Appendix B and similarly to Appendix C.

3.7. Data extraction

The data values are extracted from 238 articles in the systematic literature survey. After selecting the papers from the premier data sources mentioned in Table 4, inclusion-exclusion criteria have been applied. The quality is evaluated as per Appendix A. However some of the attributes for quality evaluation forms matched with data extraction forms. The articles are selected for the systematic review from the initial dates of the digital libraries to September 2015. The entire process of data extraction procedure is described as following:

(a) One of the authors reviewed all of the papers and extracted data from all the 238 agreed primary studies.
(b) One of the prominent professors in the field of software engineering performs the validation of the work by evaluating the papers on different parameters mentioned in the quality and data extraction forms (Appendix A–D).
(c) In case of discrepancy among the results a get-together is made to settle down the issue.

4. Code smells detection approaches

Several approaches have been introduced by the authors for uncovering the smells from the software systems. The techniques are assembled according to the type of category in which they fall. The detection methodologies vary from manual to visualization based, semi-automatic studies, automatic studies, empirical based evaluation and metric based detection of smells.

4.1. Manual detection techniques

Such detection strategies involve the human intervention for code smells detection. However it was a tedious and time consuming job for software quality maintenance. The approach followed was an initiative towards visualization based and semi-automatic approaches. It diverges from software reading method to metric based and template driven approaches. Initially, manual detection approaches were discussed by Allen and Garlan [23,24].

Noble [25] specified the relationship between different types of design patterns in the form of primary and secondary relationships. The primary relationships included – uses, refines and conflicts where the secondary relation includes used by requires, combining, etc. Travassos et al. [26] provided a manual detection strategy. It was focused on Object Oriented Systems to a software reading technique which provided assistance in detection of smells. Smith and William [27] investigated the effect of the God Class anti-pattern on the system and showed a way to avoid it.

Table 4
Databases searched.

S No.	E-resource	Search keyword	Date of search	No. of papers	Paper type
1	http://ieeexplore.ieee.org, http://www.springer.com/in, http://dl.acm.org, http://www.sciencedirect.com	Refactoring	Initial Dates-Sep 2015	85	All
		Anti-Patterns	Initial Dates-Sep 2015	25	All
		Code Smells	Initial Dates-Sep 2015	62	All
		S/W Metrics & anti-Patterns	Initial Dates-Sep 2015	21	All
		S/W Maintenance & Code Smells	Initial Dates-Sep 2015	19	All
2.	http://ieeexplore.ieee.org, http://dl.acm.org, http://www.sciencedirect.com	Object Oriented Design & Refactoring	Initial Dates-Sep 2015	17	All
3.	http://ieeexplore.ieee.org, http://www.springer.com/in, http://www.sciencedirect.com	Fault Tolerance & Object Oriented Design	Initial Dates-Sep 2015	4	All
		Anti-Patterns in Cross Company Projects	Initial Dates-Sep 2015	5	All

They proposed three new performance anti-patterns that often occur within the software systems.

Dashofy et al. [29] suggested the manual investigation of code smells in the software product. Munro [30] suggested a template driven model to detect anti-patterns. The template consisted of three components: Name of smell, description of the properties of code smell in text format, Heuristics for the detection of smells. Alikacem and Sahraoui [31] suggested a language which identified the methodology for revealing the smells in object oriented systems. The terminology provided the guidelines with the help of fuzzy logic, metrics, association and inheritance. Ghannem et al. [32] proposed a methodology to remove a bad smell from the code. Genetic Algorithms were used by the authors on four open source projects (GanttProject, Xerces-J, Quick and JHotdraw).

4.2. Visualization based techniques

In visualization based detection techniques the researchers used different approaches like Visualization orientated strategy, Visualized design defect detection strategy, Domain Specific Language etc. Some of the approaches are discussed below.

Simon et al. [33] suggested a powerful technique to inspect the internal quality of the software using a metric based visualization approach. Four types of source code refactorings had been analyzed: move function, move attribute, extract class and inline class. An enhanced metric tool Crocodile had been used. Baudry et al. [34] revealed the testing of OO system's design by calculating the design patterns and their impact on the system. To implement the works a testability grid had been proposed of design patterns and anti-patterns. Langelier et al. [35] specified a visualization approach for the quality analysis of large scale systems. A framework had been provided which was implemented on open source systems. Geometrical 3D Box was used for the representation of classes.

Binkley et al. [37] proposed a new term called Dependence Anti Pattern. Two types of techniques namely – dependence analysis and visualization analysis had been used for the recognition of anti-patterns.

Tekin et al. [42] suggested a sub-graph mining based approach to identify design patterns and anti-patterns at the designing level. The identification of such patterns in design will help the software designers to produce quality designs. Hosseini and Azgomi [43] focused on the behavior maintenance in refactoring of UML model. Author introduced control-flow diagrams (CFDs) to direct the process of model refactoring. Using a CFD for every refactoring operation, all of the requirements will be checked according to the graph grammar theory.

4.3. Semi-automatic techniques

Such detection techniques involve semi-automatic strategy for the detection of smells. Some of the work done by researchers is discussed here.

Moha et al. [39] specified a domain specific language based on DÉCOR for unmasking anti-patterns. It was a mechanism which provided a track for description of anti-patterns, by going through the sequence of steps: Description analysis, Specification, Processing, Detection, and Validation.

Noble [44] introduced a number of patterns into the pattern language based on the relationship between the patterns. Feng et al. [45] revealed that the presence of anti-patterns in the system degraded the performance of source code. A micro-architecture was proposed to generate XML based design template and Case Based Reasoning has been used which reveal the anti-patterns. Baudry et al. [46] put forward an idea of recognizing anti-patterns at the design level rather than at implementation level. UML extension method had been used to make the design better. The study was limited due to the static evaluation of phase.

Correa and Werner [48] observed that object constraint language specifications were difficult to construct under composite expressions. They further explained that in order to realize these requirements refactoring techniques were needed. Van Rompaey et al. [49] provided a view for a high-quality unit testing based on specific rules. The violation of those rules and principles were termed as test smells. The paper focused structural deficiencies of the test smells by clearly discussing their concepts and characteristics. Murphy-Hill et al. [50] proposed the techniques for refactoring like different ways to perform refactoring, generation of suppositions and finally the solutions for such suppositions. Different types of techniques like manipulating histories of code, understanding the work of Programmers and Cataloguing the use of tools have been proposed.

Nguyen and Pooley [52] exposed the presence of design patterns in UML which will be based on the accessible methods in the literature. Authors proposed two methods which will be helpful in detecting design patterns in UML. Cornelio et al. [54] suggested refactoring as transformation rules to apply on the programming languages for refactoring. Various laws and semantic rules were defined by the author and proved with example. Zhang et al. [55] performed an initial relationship between faults in software and six bad smells given by Fowler [1]. The justified relationship would be helpful in applying refactoring by the software developers. Eclipse (Core) and Apache source code were considered for the work. Fontana and Zanoni [56] identified the relationship among the code smells. Both the direct and indirect relationships were identified among the smells. Data Class, Feature Envy and God Class are selected for validating the results using the Weka Algorithm.

Shuai and Huaxin [57] projected the use of design patterns in object oriented designing which ultimately improved the code reusability. The work introduced new types of design patterns – Iterator Pattern, Adapter Pattern etc. A very sincere standard of classes – "Open Close Principle" was used for designing new patterns. Different laws, semantic rules were applied. Yamashita and Moonen [59] evaluated up to what extent bad smells are representing the maintenance of software. Four Java based software sys-

tems were placed under the observation of the experts for maintenance activity for two weeks. Liu et al. [60] believed that potential flaws in the code lead to bad smells which were difficult to detect and remove. They reached at an analysis that up to 17.64–20% efforts can be reduced if these bad smells are detected and resolved using the proposed solution. Cortellessa et al. [61] proposed first order logic approach for revealing the response from the performance anti-patterns in the system. A rule-engine was designed which detected three performance anti-patterns – Blob, Concurrent Processing Systems, and Empty Semi Trucks.

Li and Thompson [62] introduced a Rule-based structural tool that allowed the user to perform the refactoring from scratch. The tool was an advancement of already existing tool Wrangler designed for the language Erlang. Future work included the refactoring of Haskell programs. Murphy-Hill and Black [63] proposed refactoring tools that display an error message in which suggestions for correcting the problem were mentioned. Liu et al. [64] proposed a new kind of semi-automatic generalization refactoring approach where the classes and interfaces supporting inheritance were managed. The work introduced a tool called GenReferee (Generalization Referee) to disclose the inheritance dependencies which selected the class's pair wise. Fontana et al. [65] proposed a new methodology of machine learning algorithms to identify the bad smells in the software's code. An enormous size of data set was selected to run more than 15 machine learning algorithms to detect the code smells. An accuracy of 90% was provided by most of the algorithms for the smells.

Mahmoud and Niu [66] presented three methods for refactoring a system. First method was Rename identifier which was very beneficial in improving traceability. The second was the extract method where similar code repeated again had been removed. The third was Move method, in which the code placed at inappropriate location, has been shifted to its correct place. Kessentini et al. [67] proposed that software refactoring was one of the best methods to remove the problems from coding and to maintain the quality of software. Author stated that problems from the code can be eradicated by using properly designed code.

Leitao [68] proposed a template matching approach for pattern languages that would help in performing refactoring of Lisp Programs. The implementation was used at a refactoring tool C^3PO (Careful Code Conversion to Prevent Obsolescence) which was used for refactoring legacy system SNePS. Hegedus [69] revealed that certain coding activities lead to distraction of maintainability like applying refactoring, patterns and anti-patterns detection. The performance of reverse engineering tools was evaluated so as to improve the maintenance. Wang et al. [70] suggested a new incremental approach of refactoring called RINV (every function has a right to inverse) using which refactoring can be performed in an easier manner. A framework was developed which shifts the source code into proper encapsulated code. Ujhelyi et al. [71] suggested that anti-patterns could be identified in the system using queries and approaches like local search approach and incremental approach. Pattern matching approach was declared more absolute than hand coded approach.

Yamashita [74] identified negative impact of more than ten code smells on the maintenance activity. Six developers were asked to maintain four medium sized applications for six weeks. Feedback was taken from the observers on daily basis and concluded that contravention of Interface Segregation Principle is done. Kessentini et al. [75] used a search based approach based on Parallel Evolutionary algorithms to detect the code smells parallel in all the methods. The approach was more accurate than the other detection methods. Khan and El-Attar [76] suggested the model transformation approach for refactoring of the use case models so as to enhance the quality of software products. MAP-STEDI use case model was considered for the removal of anti-patterns and refactoring. Unterholzner [77] introduced the improvement needed for the tools of refactoring in Smalltalk programming language. The static type systems (compile time) were judged by the tools not the dynamic one (run time). The precision of the tool is more than 45%. Tilevich and Smaragdakis [81] suggested a new type of refactoring called Binary Refactoring in OO systems. The target of refactoring was to remember the changes done after refactoring. A binary catalogue was proposed that express the effectiveness of refactoring. A Java browser BARBER (author's previous work) had been taken into consideration.

Wohlfarth and Riebisch [82] needed to facilitate Architecture-oriented refactoring. By this adaptation the structure of the architecture oriented refactoring process could be improved. The application of the method was demonstrated using examples from a case study. Mealy and Stropper [83] used DESMET Feature Analysis technique for assessing refactoring techniques. It was used to assess the productivity of six java refactoring tools. To weigh the categories on the basis of their importance was the major feature of the standard DESMET Feature analysis method. Piveta et al. [84] have described an approach for reducing the search space for refactoring opportunities. It provided mechanisms for creating and simplifying a DFA representing the applicable refactoring sequences in existing software. Meananeatra and Rongviriyapanish [86] introduced refactoring filtering conditions, which help novice programmers for finding candidate refactoring. These conditions have been defined at the element level. Abdel-Hamid and Noaman [87] presented more systematic approach to refactoring that they have found to be successful for refactoring legacy code. This report described two experiences that one with 3 teams applies a basic and traditional refactoring approach and another with 2 teams applying the new approach. Wongpiang and Muenchaisri [90] proposed an approach for selecting the sequence of refactoring techniques usage for code changing based on Greedy Algorithm. The approach was evaluated with source code containing Long Method, Large Class and Feature Envy bad smell. The compared results showed that changed source code (after applying refactoring) improved software maintainability. Chu et al. [91] revealed test case refactoring for Extreme Programming which leads to faster development. Two patterns from the Gang of Four were selected for the application of refactoring and the validation of study.

Bansiya and Davis [95] introduced a hierarchical prototype for the evaluation of quality attributes (reusability, flexibility, understandability, etc.) in object oriented designs. Gueheneuc and Antoniol [99] presented Design Motif Identification Multilayered Approach (DeMIMA) for the detection of micro architectures (complementary to design motifs). It was a three layered architecture in which the first two layers provided a miniature of the source code, and the third layer identified design patterns. Khomh and Vaucher [106] proposed Bayesian Detection Expert, a Goal Question Metric (GQM) based approach to construct Bayesian Belief Networks (BBNs) from the descriptions of anti-patterns. BDTEX was validated for three anti-patterns: Blob, Functional Decomposition, and Spaghetti code on two open source systems Gantt Project and Xerces. The approach was again verified on two industrial projects Eclipse & JHotDraw. Palomba et al. [109] investigated different versions of the software to identify the code smells and comparing the versions for the number of smells. The approach was named as HIST for revealing the smells – Divergent Change, Shotgun Surgery, Parallel Inheritance, Blob, and Feature Envy. Maiga et al. [126] proposed a machine learning technique called SVMDetect which automatically detect anti-patterns in the system. The implementation and analysis of Blob Class had been done using SVMDetect and DETEX on ArgoUML, Azureus and Xerces. The author revealed that the accuracy of SVMDetect was better than DETEX. Sjoberg et al. [130] tried to build a relationship between the more than ten code smells and maintenance of the software product. Four Java systems

were kept under the observation of six experts for their maintenance activity. InCode and Borland Together were considered to detect bad smells. Pandiyavathi and Manochandar [152] suggested the methods for code smells in a system. An overview of refactoring technique was proposed which would be time saving. An algorithm was proposed to implement the refactoring methods. Palma et al. [153] specified the detection of anti-patterns in business processes. The rule-based approach was detected for improving the quality of BPEL (Business Process Execution Language) processes to detect BP anti-patterns. Palomba [161] discussed how the smells can be detected using the text based conceptual information. The term TACO (Textual Analysis for Code smell detectiOn) has been introduced by the author to identify Long Method smell on three Java projects. Ujhelyi [162] exposed different query based anti-pattern detection techniques for the performance estimation. For this purpose more than 25 Java projects were evaluated. Refactoring was applied for detecting the anti-patterns using an incremental, local search based policy. Briand et al. [174] introduced a novel approach to measure the accuracy of fault identification models. A technique was introduced by the author called MARS (Multivariate Adaptive Regression Splines) which will be helped for generating such models. Thus the novelty of such models was evaluated through the work Kruchten [181] exposed the actual work done by the software architectures for maintenance activity. The study proposed the name architectural anti-patterns which if present in the code leads to difficulties for the software architectures. Luo et al. [187] introduced the new model for software maintainability i.e. connection between the code smells and anti-patterns. The author stated that quality could be improved by minimizing the affect of code smells on the code.

4.4. Automatic detection techniques

In case of automatic detection strategies models, frameworks, surveys and generation of fully automatic detection tools has been considered. Different types of code smells are identified and the approaches are validated on real world systems. Some of the approaches are discussed here.

Astels [10] performed the refactoring of UML diagrams. The approach could be beneficial for designing reverse engineering tools. Thus refactoring was performed by transforming the diagrams into the equivalent source code. Olbrich et al. [15] evaluated the destructive nature of bad smells using two kinds of classes-God Class and Brain Class. Three open source systems – Log4j, Apache Lucene, and Apache Xerces were examined. A tool was introduced called- EvolutionAnalyzer for the detection of smells. Liu et al. [18] introduced monitor supported refactoring framework that permitted automatic refactoring of code which provided benefits to the novice programmers. The detected methodology was compared with other refactoring tools and provided an approximately accuracy of 50%.

Ganea et al. [20] introduced an Eclipse plug-in called "InCode" which is beneficial for providing high quality code solutions to Java programs. Code smells like Code Duplication, God Class, Feature Envy and Data Class could be easily located by InCode. For the validation of results open-source system JFreeChart was selected by more than eight industry experts.

Murphy-Hill and Black [40] believed that code smells were the indicator of design defects. Smell detectors acted as a warning for the programmers to alert them about the presence of these errors. The authors discussed a smell detector called Stench Blossom which helps the programmers to give the overview of the smells.

Bryton et al. [41] suggested that the implementation of refactoring is practical in nature. The detection of the Long Method smell was implemented using Binary logistic regression model. Ballis et al. [47] provided a sound tool with strong GUI for finding the patterns and anti-patterns in UML. Different types of anti-patterns were investigated in the relevant examples. The suggested approach could improve the capabilities of the class diagrams. Kempf et al. [51] suggested a new approach for performing cross language refactoring using the eclipse plug-in. Groovy language was validated.

Vidal et al. [73] presented a novel approach to schedule the code smells in performing refactoring. A tool was introduced that focused on various levels. Two case studies had been validated for the proposed tool.

Gatrell and Counsell [80] performed the refactoring of large size C# software to identify the faults and changes that were made after refactoring. Bespoke tool has been used to identify 15 types of refactoring. Gueheneuc et al. [92] classified three types of design defects i.e. Intra-Class (within class), Inter-Class (among classes) and Semantic Nature. A Meta model had been used to describe design patterns. Tokuda and Batory [93] disclosed that design patterns were interrelated with the refactoring in OO programs. Three types of design originations were disclosed usual with refactoring and managed with the GUI developed by the authors. C++ code is managed by the automated tool. Ekman and Schafer [94] extended JastAdd (compiler) by generating an engine that automatically performed refactoring. The work shall be used as base for other types of refactoring like Extract Method, Rename Component, etc.

Gueheneuc [96] introduced a tool suite "Ptidej" which provided the capability to reverse-engineer different programming languages to UML class diagrams accurately. PTIDEJ generated the UML class diagrams which helped in identification of code smells with a higher level of abstraction.

Washizaki and Fukazawa [97] defined a new refactoring approach – Extract Component which pulled out the reclaim element from the object oriented programs. A Java parser generator JavaCC has been used. The proposed system performs automatic refactoring of Java programs. Llano and Pooley [101] proposed a guide to the automatic detection and correction of OO anti-patterns in UML. Hassaine et al. [102] introduced IDS – a system that identify the presence of code smells and anti-patterns. Gantt Project v1.10.2 and Xerces v2.7.0 were checked for the existence of smells. Fontana and Zanoni [103] proposed a tool called MARPLE used for the detection of patterns. The tool was used as an Eclipse plug-in. Different components of MARPLE were introduced like Information detector engine module, Joiner module and Classifier module.

Chatzigeorgiou and Manakos [104] examined OOP systems for the detection of smells. Two open source systems (JFlex and JFreeChart) have been taken into consideration for multiple versions and four different types of smells are detected using jDeodorant.

Tsantalis and Chatzigeorgiou [105] highlighted a method of performing refactoring in Java so as to introduce polymorphism. The suggested idea was implemented in the form of an eclipse plug-in. Open source projects – UML Editor Violet 0.16, Ice Hockey Manager 0.1. And Nutch 0.4 were used in the study for the validation. Fokaefs et al. [108] suggested automatic refactoring of extract method based on complete computation slice and object state slice. The proposed work identify the design defects which affected coupling and cohesion in the form of Eclipse plug-in. Feature Envy Bad smell was detected using the refactoring procedure. Both the independent and dependent refactoring was performed on open source software JFreeChart. Abebe et al. [110] revealed that maintenance activity was highly influenced by lexical bad smells. Techniques were applied before and after the refactoring and the outcomes are validated. LBSDetectors tool had been used in the inspection. Borg and Kropp [111] introduced a tool ReFIT (Eclipse plug-in) that automatically performed the maintenance at testing level based on refactoring approach. The tool is freely available on SourceForge.

Maruyama and Omori [112] introduced a fresh idea by developing the tool Jsart (Java security-aware refactoring tool). It automat-

ically performs refactoring by concerning the issue of software security. Oliveira et al. [113] discussed Reuse Tool, to facilitate the instantiation of Object Oriented Frameworks. Fontana et al. [115] proposed code smell detection tools available in the market. Six versions of Gantt Project were explored for the detection of four types of code smell, using more than six tools. Maiga et al. [118] described "SMURF" which is an Anti-pattern Detection Approach. More than 290 experiments have been conducted on three systems i.e. ArgoUML, Xerces, Azureus. Four types of anti-patterns Blob, Spaghetti Code, Functional Decomposition, and Swiss Army Knife were identified. Author revealed that the accuracy rate of SMURF was greater than that of DETEX and BDTEX for detection of anti-patterns in the system. Kaur and Singh [119] carried out the detection of type checking code smell using Eclispe plugin jDeodorant. The source code of jHotDraw was taken as input. Brown et al. [120] suggested an innovative approach of refactoring applied on different languages like C, C++, Haskell, Python and Erlang. Schafer [121] proposed the idea of automatic approach refactoring of dynamic languages. The challenges and work done by researchers in the field of dynamic languages were discussed. Christopoulou et al. [122] aimed at applying automatic refactoring for the removal of bad smells which included difficulties of multifarious conditional statements. The algorithm automatically detected and removed the code flaws using Eclipse plug-in JDeodorant. Perez and Crespo [123] presented an approach to perform refactoring in an easier manner. The planning known as HTN was used for planning of refactoring before actual refactoring. Einarsson and Neukirchen [124] suggested an idea to transform UML diagrams and models into the refactored one. The main advantage of their technique was that it could easily perform the refactoring of both the model and the UML diagrams as well. For the verification of work an Eclipse plug-in Papyrus UML Editor was implemented.

Peters and Zaidman [125] assessed the duration of a code smell during the lifetimes of different versions. SACSEA tool was developed for code smells detection in Java. Nguyen et al. [127] proposed a new class of smell called "embedded code smells". Most of the web applications suffered from embedded code smells as the applications are server side. The author proposed a tool called WebScent to detect the smells. Danphitsanuphan and Suwantada [128] proposed a tool called BSDT (Bad Smell Detection Tool) which was an Eclipse plug-in used for revealing bad smells in code. Different types of bad smells were detected from the java source codes using Naive Bayes Probability Estimation. Han and Bae [129] proposed an automatic approach to uncover the expensive refactoring in OO systems. The dynamic profiling method was used to identify the finest refactoring. Three open source projects (jEdit, Columba and JGIT) were scrutinized under dynamic profiling.

Chatzigeorgiou and Manakos [132] identify the bad design practices in two open source systems (JFlex, JFreeChart). Four smells – Long Method, Feature Envy, State Checking and God Class were identified using jDeodorant (an Eclipse plug-in). A survival analysis was done that revealed, once a smell entered into software it remained there as a fixed entity. Fontana et al. [133] performed the study to identify the influence of code smells on the quality of software in different domains like client server software, application software etc. NiCad, iPlasma and Excel macros tools were used to detect the smells. The most commonly occurring smells were Duplicate Code, Data Class, God Class, Schizophrenic Class and Long Method. Ligu et al. [134] suggested that Refused Bequest was a very common coding problem in object oriented programming. Author proposed a method of unit testing after introducing some errors explicitly in the code, in order to check whether the inherited methods are employed. Palomba et al. [135] revealed that poor programming leads to problems in system known as code smell. Code smells could be identified with the help of many techniques. HIST (Historical Information for smell detection) was one of the basic and famous techniques with an accuracy rate from 61% to 100%. Soares et al. [136] performed the testing of Java refactoring tools to study their behavior using the tool SAFEREFACTOR. Arendt and Taentzer [137] suggested a static quality structure which was based on metrics, thus providing automatic tool support for performing refactoring. The tool was a multitasking environment that identified code smells, metric values and refactoring based on Eclipse Platform.

Bavota et al. [138] introduced the Extract Class refactoring to identify the methods that are strongly related (high cohesion). The study was an extension of author's previous work. Kim et al. [139] proposed the automatic detection of bad smells in the code using the OCL (Object Constraint Language) based on JavaEAST model. The model was verified by running on an Eclipse plug-into detect the smells. Orchard and Rice [141] performed the automatic refactoring of Fortan language using the tool CamFort. Jiau et al. [142] proposed a new methodology OBEY for optimal batch refactoring plan at three levels. The work revealed Middle Man Refactoring in three open source software systems JDTCore, HSQLDB and JEdit.

Lakshmanan and Manikandan [145] revealed the detection and correction of bad smells (refactoring) in the code. Four types of smells-Duplicated Code, Long Method, Large Class and Long Parameter List were focused. The tool named JDEv was applied on the source code to detect the smells.

Kumar and Chhabra [146] presented a two phase dynamic methodology to detect the Feature Envy smell. The first phase detected smell using jProfiler and the second used AspectJ to rectify such smells. Two open source projects Eclipse and jEdit 3-5.1.0 2 (only one package) were used to verify the result. Bavota et al. [147] aimed at removing Feature Envy Bad Smell from the code using Move Method Refactoring. The work done was a two step progression. Six projects – AgilePlanner, eXVantage, JFreeChart, GESA, SMOS, and JEdit have been considered for design defect removal. The performance of the proposed methodology was compared with Eclipse plug-in jDeodarant. Chaudron et al. [148] focused on the design quality by developing the tool – PoSDef. Two open source projects – Movie Maniac and College Chef were investigated with one industrial project. Thus, PoSDef was beneficial for the software designers as they could target more clearly on its suggested outputs. Peiris and Hill [149] brought up the idea of non-intrusive performance anti-patterns detector (NiPAD) which was categorized into two levels. The first level was Training Level and the second was Prediction Level. Five types of classifiers were considered- Logistic, FLD, Naive Bayes, SVM, SVM (RBF) which tested for sensitivity, specificity and accuracy. Jaafar et al. [150] investigated three Java based systems to identify the transformations of anti-patterns in all the previous versions of the software for more than 10 code smells. The study revealed that anti-patterns transformed from one type to another throughout its life cycle.

Mongiovi et al. [151] suggested a novel approach called Safer Refactoring using the tool "SafeRefactorImpact". It was able to detect the methods containing alteration using analyzer in either OO or AO systems. Behavioral changes were accurately identified by the approach. He concluded that SafeRefactorImpact was more precise than the previous work.

Palma and An [154] proposed that the quality of service based systems get affected by the use of anti-patterns. Based on the data collected from the SBS FraSCAti, it was shown that the services suspicious of anti-patterns require more maintenance that non patterns services.

Hall and Zang [156] suggested a tool for the automatic detection of five code smells proposed by Fowler. Three open source projects – Eclipse, ArgoUML, and Apache Commons were taken for validation. The source repositories of the projects were scanned to iden-

tify the changes and faults in the systems. Han et al. [158] suggested an automatic detection of move method refactoring opportunities to provide the excellent maintenance of software system. Maximal independent set (MIS) was anticipated which detected the maximum number of refactoring. Three open source projects (Edit, Columba, and JGit) were taken into account for multiple refactoring. Gaitani et al. [159] proposed automatic detection and correction of null object design pattern. An algorithm was introduced in the form of Eclipse Plug-in for various open source systems such as violet, nutch, myfaces-impl, jackrabbit-core, apache ant, jade, xerces, jfreechart and batik. Ammar and Bhiri [160] suggested a novel approach of refactoring in collaboration with UML, B and CSP which could be applied at the advanced level of software development. Class diagram have been deemed for the validation Software Structures Analysis Tool's. The work aided in identifying the problematic areas and correcting them.

Morales [163] brought the idea of developing framework that will correct anti-patterns automatically. A search based refactoring was applied, for the removal of anti-pattern on Eclipse plug-in. Liu et al. [164] disclosed about the Rename Refactoring possibilities but did not involve complex semantic investigation. The refactoring applied changed the name of entities that were given wrong names. Rename Expander (Eclipse Plug-in) was developed for Eclipse IDE to detect Rename Refactoring. Nongpong [165] proposed a metric to detect the feature envy class called "Feature Envy Detector". The move method and extract method refactoring were suggested as one possible solution. JCodeCanine tool was proposed as an Eclipse plug-in for the verification of metric on small sized OO applications.

Mancl [166] introduced refactoring which lead to reusability of software. The author focused on applying refactoring at different stages and testing each stage. Xing and Stroulia [167] scrutinized three pairs of Eclipse versions and revealed though refactoring was a common practice which involved diverse restructuring types. Lan et al. [168] revealed a middleware based approach for refactoring. Meta model architecture was used for differentiating the good and bad patterns that would perform refactoring automatically.

Maruyama and Takoda [169] proposed security-aware refactoring which gave the information on the decreasing of security level of modified program. They developed a tool which detected the decreasing of access levels of fields in two kinds of security-aware refactoring. Sondergaard et al. [170] presented the refactoring of the Real-Time Specification for Java (RTSJ) which highlighted core concepts. It was a suitable foundation for the proposed levels of the Safety Critical Java (SCJ) profile (JSR-302).

4.5. Empirical detection techniques

The following are the empirical detection techniques which explore the empirical studies, the models generated and the surveys performed on code smells and refactoring. Different types of methods have been considered by different authors. Some of the proposed work is given below.

Al Dallal [2] performed a systematic literature survey on the refactoring process in object oriented systems which provided an opening to the actions for refactoring. The author accomplished that a high rate of work was taken under consideration on open source systems with the same repeatable datasets. The commonly used refactoring was Extract Class and Move Method. Six types of categories were specified by the author for performing refactoring.

Zhang and Hall [4] provided a deep insight of literature by going through more than 300 papers on code bad smells since 2000. The author suggested that research work was needed on the percussion of code smells. It had been concluded that the smell-Duplicated Code was studied more than other code smells.

Mens and Tourwe [5] performed a survey on refactoring of software systems including the techniques for refactoring, tools for refactoring, software on which refactoring is performed and how the refactoring improves the quality. Different types of refactoring have been performed on an OO system.

Tufano et al. [17] revealed the reason for occurring code smells and their location of presence in the code. More than 200 open source projects were investigated with their versions to uncover the problem of bad smells. Abebe and Yoo [53] performed the systematic literature survey on the term refactoring. The purpose was to grasp the refactoring knowledge through categorization and summation.

Ghannem et al. [79] aimed at investigating the design imperfections and measure likeness between design flaws and software under consideration. Four open source systems – GanttProject, LOG4J, ArgoUML and Xerces were studied to uncover the Blob and Functional Decomposition.

Counsell and Hassoun [98] described the refactoring of seven open source Java systems – MegaMek, JasperReports, Antlr, Tyrant, PDFBox, Velocity and HSQLDB. The results demonstrated that most common re-engineering components of open source systems are – Renaming and Moving fields/methods among the code. Vaucher et al. [100] examined carefully the God Classes to detect the occurrence of bad smells in the software. Xerces and Eclipse JDT (open-source systems) – had been studied for the investigation of God Classes. Singh and Kahlon [114] introduced a metric model for investigating the classes with code smells in the system. The results obtained from metrics could be helpful in determining the code smells and faulty classes.

Romano and Raila [116] studied the system by considering source code changes (SCC) obtaining from 16 Java open source systems. Three anti-patterns Complex Class, Spaghetti Code, and Swiss Army Knife were identified. Khomh et al. [117] investigated the affect of anti-patterns on classes. More than 50 releases of four systems – ArgoUML, Eclipse, Mylyn, and Rhino had been considered. Thirteen types of anti-patterns were identified. The relation between the anti-patterns with the change-tendency and anti-patterns with fault-tendency was investigated. Jaafar et al. [143] provided a relationship between anti-patterns and design patterns. Three open source systems ArgoUML, JFreeChart and XercesJ were considered for the evaluation of relationship. It was concluded that relationship exists between anti-patterns and design patterns but on temporary basis. The classes had more error tendency which was present in such anti-patterns. Dexun et al. [157] suggested that the classes which were functionally not related could generate problems in software maintenance. Hence the detection and refactoring of such classes was needed. A bad smell was proposed by the authors named – Functional over related classes (FRC). A detection strategy was suggested to indentify the bad smell. The work was validated on four open source systems- HSQLDB, Tyrant, ArgoUML and JfreeChart.

Briand et al. [173] performed a case study to explore the term quality in object oriented designed systems with coupling and cohesion. The study was a replica of the author's previous paper where the work was validated by the students on minor projects. Now the proposed work was undertaken by industry professionals justified in [38].

Mantyla et al. [175] provided a categorization of code smells with an arrangement. The author mentioned classes for smells like bloaters, object-orientation abusers, change preventers, couplers, etc. The nomenclature presented a relationship between the smells.

Mens et al. [176] proposed an introduction about refactoring from different perspectives. Various research questions were answered like how the tools should be developed, association among quality and refactoring, when and where to apply refactor-

ing, etc. The author mentioned the need of automatic refactoring tools for flexible refactoring as the future direction. Fabry and Mens [177] proposed a Meta level interface which will detect the complex details about the source code. Two applications coded in Smalltalk and Java language were used for the interface using logic queries. The results were manually validated for the source codes. Khaer et al. [179] revealed that the software quality was highly influenced by design patterns. An empirical investigation was done to calculate the software quality. For the valuation of the concept GoF (Gang of four) design patterns were selected. PINOT tool was selected to validate the results.

Li and Shatnawi [180] performed an empirical investigation for the association of smells with faults in the code of post release OO software. The study was carried out on a famous open source system called Eclipse for three releases. Bugzilla was used to verify the results collected, the errors removed and with what type of solution. Khomh and Gael [182] introduced a concept of software quality maintenance by avoiding the use of harmful anti-patterns. Olbrich and Cruzes [183] considered the historical data of Lucene and Xerces. It had been identified that classes with the Blob and Shotgun Surgery anti-patterns had a higher change frequency than non anti-patterns classes.

Carneiro et al. [185] presented a compound apprehension from different perspectives to recognize the bad smells in code. The author concluded that the concern driven perspective was useful to identify God Class.

Bertran et al. [186] suggested that aspect oriented programming improved software maintainability but, it also added new errors in the program which are termed as code smells. The paper gave the detailed analysis of code smells which were involved in developing aspect oriented systems. Fontana et al. [188] explored different tools for the detection of smells in the code and revealed how these tools were different from one another. PMD, iPlasma, InFusion, StenchBlossom, DECOR (Black Box) and JDeodorant were the few tools introduced by the author. Most of the smell detection tools targeted Java language. Macia et al. [189] performed an exploratory study by detecting the code smells in AOP (Aspect Oriented Programming) and how they influence the modularity of such systems. Versions of two software designed using different architectures were selected for the detection of smells. Singh and Kahlon [190] developed a model which determined the faulty classes in the system. A binary statistical relationship has been created between code smells and software metrics. The bad smells identified in the code were categorized into different types. Two types of categories – Bloater and Change Preventer were examined carefully by the developed model. Weber et al. [191] performed a survey on refactoring of large repositories. A catalogue of smells was defined by the authors for refactoring. Two projects on healthcare and automotive spheres were evaluated for more than 10 types of refactoring. The suggested work helped software community for managing process repositories. Izurieta and Bieman [193] explored different reasons for crumbling of design with the passage of time. Three object oriented systems were taken into consideration for the research. Ouni et al. [195] provided Genetic Programming Approach which demanded two steps i.e. automatic detection and correction of maintenance problems. Blob, Sphagetti Code and Functional Decomposition were detected on six open source projects Xerces-J, ArgoUML, Log4j, Azureus, Quick UML and Gantt.

Dersten et al. [196] performed a case study to identify the refactoring likelihood in the embedded systems. Group Interviews were performed with the architects and software experts for improvement of quality using refactoring. The author concluded that same refactoring guideline could be applied to any type of products. Kim et al. [197] analyzed the con-front and advantages of performing refactoring. The author performed a field investigation at Microsoft with survey, interview with professionals and scrutiny of Windows 7 operating system.

Yamashita and Counsell [198] justified the fact that presence of smells in the code was automatic or it was due to unawareness of the developers about smells. A case study was performed with open ended questionnaire to be answered by 85 software experts. Lack to appropriate tools for smell detection was a major concern. Pinto and Kamei [199] performed an investigation for the review about the refactoring tools form the StackOverFlow Society. Various research questions were designed and answers were noted with the percentage of views. The advantages and needs of refactoring tools were countered by industry experts. Balaban et al. [200] provided a quality model detected the problems in class diagrams by using a catalogue of anti-patterns for maintenance. The author explored that correcting the anti-patterns in the code lead to the quality improvement of the product.

Misbhauddin and Alshayeb [201] discussed the various refactoring models which were related to the field of unified modeling language. A review of 3295 articles was made by them. The author summarized that UML model refactoring was a significant area of research. Singh and Mittal [202] introduced an empirical model for calculating the faults in Firefox Mozilla using metrics. Three categories of bugs were reported- High, Medium and Low-impact. Columbus Wrapper Framework Tool was used calculating the results from object oriented metrics.

Wert et al. [204] discussed various heuristics measures for detecting the performance of anti patterns. These heuristics were evaluated based on TPC-W. Sahin et al. [205] identify the code smells at two levels where the first level uncovered the detection rules and the next disclosed the smells that remained silent during the first level. A set of nine open source systems and one industrial project was taken as dataset and the accuracy was more than 85%. Abebe and Yoo [208] performed a literature survey to identify the research issues in refactoring. The author summarized the research papers from the year 1999 onwards to 2014. More than 150 papers are studied and suggestions were recommended with future work. Morales et al. [209] investigated the relation between code review applications and the influence on the superiority of design. Seven types of anti-patterns were selected to justify the poorness of design and its affect on the quality. Jaafar et al. [210] studied the influence of anti-patterns and the faults introduced by them. It was explored that with the change in version of software systems, the anti-patterns change themselves. ArgoUML, JFreeChart, and XercesJ were selected to identify design patterns and anti-patterns with more than 35 releases.

Fenske and Sehulze [211] introduced four kinds of variability-aware code smells which were detrimental for the software code. It was concluded that code smells in the systems further leads to the difficulty in maintenance and understanding. Abílio et al. [212] discussed the presence of three code smells- God Method, God Class, and Shotgun Surgery in AHEAD (a powerful FOP language) and investigated the systems using proposed detection methodology. Palomba et al. [213] revealed that there was an absence of techniques in the literature for evaluating code smells. Thus the author introduced LANDFILL which was an online platform to discuss the bad smells data suite.

Bavota et al. [214] made the users aware about smells during unit testing called "Test Smells" also known as bad test code smells. Two types of investigations have been performed. The study revealed that both the open source and industrial products contain at least one bad smell with harmful effects on the system. Hence difficulties were faced by the testers in maintenance phase. Ouni et al. [215] revealed the best practices for performing refactoring which reduced the chances of getting smelly classes. Four types of smells were evaluated on five software systems – GanttProject, JHotDraw, ArtOflllusion, JFreeChart and Xerces-J.

Kaur and Singh [216] performed a vast literature survey to identify different approaches for the detection of code smells and anti-patterns. The study provided four broad categories for the approaches followed by the researchers in the field of bad smell detection. Chen et al. [217] introduced Iterative development model which was used to refactor the software according to the present needs of the user. The author presented survey on refactoring practices which indicated the demand of research for the methodologies that are needed to be used for refactoring in future. Neto et al. [218] suggested the areas to apply refactoring. A platform was provided called "AutoRefactoring" which revealed agents that manage most of the refactoring problems. Five Java open source projects with different versions, namely Log4j, SweetHome3D, HSQLDB, jEdit8 and Xerces were considered.

Khomh et al. [219] investigated the influence of bad smells in code with respect to change proneness. Two open source projects – Azureus and Eclipse were investigated for more than twenty versions and nine code smells were detected. It was concluded that classes containing code smells reluctant to changes as compare to other classes. Counsell [220] examined testing categories which were discussed by van Deursen and Moonen. He further explored the relationship between refactoring techniques which were suggested by Fowler.

Mealy and Stropper [83] suggested the usability of refactoring tools. He analyzed ISO 9241-11 to study the environment in which refactoring prevails. In order to study the usability of refactoring they condense 120 guidelines from 11 different sources into a list of 29 guidelines. Anbalagan and Xie [222] discussed a new modularization approach of software systems called the Aspect-Oriented Programming (AOP). He proposed two concepts of aspect mining and aspect refactoring. Aspect refactoring grouped similar join points together.

Marticorena et al. [223] presented an evaluation of the behavior of refactoring tools on the source code that defines or uses generics. Comparisons were made on the behavior of five refactoring tools on a well known refactoring of Extract Method, and its implementation for the Java language. Rizvi and Khanam [224] adopted a systematic approach to refactoring legacy software. This methodology shall be used by the developers for refactoring the legacy code in a systematic manner.

Lopez-Herrejon et al. [225] presented their experience in refactoring features based on the requirements specifications of small and medium size systems. Their work identified eight refactoring patterns that described how to extract the elements of features.

Lee [226] conducted the case study to show why and how programmers use refactoring on the code written in functional programming languages such as Haskell. He found a total of 143 refactoring classified by 12 refactoring types. Halim and Mursanto [227] noticed that the refactoring rules effect on class cohesion showed no consistent trends. There were fifteen refactoring rules that applied in fifty-three measurements.

Nguyen et al. [228] presented the empirical study on several PHP based Web applications. They developed WebDyn, a tool to support dynamicalization refactoring.

Kannangara and Wijayanayake [229] aimed to assess the impact of refactoring on code quality improvement in software maintenance. Experimental research approach was used and ten selected refactoring techniques were used. Szoke et al. [230] studied hundreds of refactoring commits from the refactoring period of five large-scale industrial systems developed by two companies. They found interesting observations based on what and how developers refactored in the project. Zhao et al. [231] provided an extensive overview of existing research in the domain of parallel refactoring. Challenges of modifying a sequential program to parallel code were reported relating to the developer, the methods and tools. Bois et al. [232] explored refactoring in practical usage of industry and research point of view. The research trends were discussed for real time embedded and safety-critical applications in the current scenario. Various automatic (Guru, Daikon) and semi automatic (Refactoring Browser, XRefactory and jFactor) tools were discussed. Santos [19] revealed that in order to understand the problem of god class code smell empirical studies were required. Studies show that human role in detection of god class proved to be relevant.

4.6. Metric based detection techniques

The metric based detection strategy includes the usage of metrics for different aspects of refactoring and revealing code smells in the code. However authors used metrics for measuring the internal quality of the product and comparing with the flaws encountered in the external code. Some of the approaches provided by researchers using metrics are discussed below.

Marinescu [28] introduced a metric based approach for detection of anti-patterns. The technique was realized on Iplasma tool. More than 8 anti-patterns were detected with nearly same number of techniques.

Alshayeb [85] investigated the effect of refactoring on the different cohesion metrics. Future research work can test more systems and bigger datasets in similar and different contexts to further confirm the results of this empirical study. Singh and Kahlon [107] investigated the importance of software metrics and encapsulation for revealing the code smells. A software metric model was introduced that provided the categorization of smells in the code. Firefox open source system was investigated for the validation of results.

Singh and Kahlon [155] revealed the importance of metrics and the threshold values in software quality assurance. Analysis of risk in software systems was explored against the threshold values for the detection of bad smells. The study was validated on the source code of three versions of Mozilla Firefox. Fontana et al. [194] identify the impact of bad smells on the quality of software. Three bad smells were identified using different object oriented software. The author depicted the change in metrics value upon the eradication of smells. Mansoor et al. [206] proposed the identification of bad smells and rejecting those parts of design that were good enough. Multi-Objective Genetic Programming was used to specify the metrics values for revealing bad smells. Shrivastava and Shrivastava [233] performed a case study to evaluate how the quality of software got influenced by software metrics. An open source project Inventor Deluxe v 1.03 was evaluated on Eclipse plug-in (Metrics 1.3.6). Basili et al. [234] discussed that the quality assurance of the object oriented software revolved around the design metrics. Chidamber and Kemerer's design metrics were extracted from the source using GEN++. A total of five metrics proved to be reflecting the fault proneness of the software.

Huston [235] revealed that internal quality of the software is measured using different types of metrics and based on their values appropriate refactoring possibilities are applied. The author studied the dependency of design patterns on design metrics. Three patterns Mediator, Visitor and Bridge were studied with non patterns forms. Mantyla [236] shared his experience on the application of refactoring in the software industry. JFactor tool was used to detect the smells and the metric CCCC verified the refactoring process. Gustafsson et al. [237] discussed that the quality of software was influenced by the software metrics. For that purpose Maisa (Metrics for Analysis and Improvement of Software Architectures) and Columbus tools were used.

Baldassari et al. [238] mentioned that if the software product was managed at the initial stage then efforts were reduced the overhead at the later on phases of the software. Chidamber and

Kemerer design metrics were used. Software named "UML Checker" was used to validate the work.

Srivisut and Muenchaisri [239] discussed a new programming paradigm called "Aspect Oriented Programming". The authors introduced AO metrics which helped in detecting flaws in code Fifteen AO metrics were used for four categories of smells. Catal et al. [240] disclosed that the fault lenience was reliant on the metrics, size of the program and feature selection techniques. Datasets were taken from NASA. Ten metrics were identified from a set of nine fault prediction algorithms. Macia et al. [221] proposed metrics based technique to disclose the bad code smells in Aspect Oriented Programming (AOP). Three AOP systems with 17 versions were validated using the approach. A high accuracy output was achieved using the strategy.

Mittal et al. [207] developed a model which determined how the object oriented metrics were used to calculate the faultiness in the classes of post-release object oriented systems. Three versions of Javassist project were evaluated for errors using the FindBug Tool. The author concluded that the model developed could be used for forecasting errors in a given system.

Dijkman et al. [203] proposed a metric based automatic detection of refactoring possibilities in the code. A large set of anti-patterns were detected and removed by refactoring. Fontana and Spinelli [192] investigated relationship between metrics and software quality after the refactoring methods had been applied. It was an interactive process where one smell eradicated from the source code reflected the change in the metric value. Three refactoring tools- JDeodorant, PMD and infusion were applied on four kinds of smells. Singh and Kahlon [184] came up with a new idea of refactoring model which targeted the code smells in Firefox. The author validated the relation between software metrics and code smells. A metric model was developed to depict the relation between metrics and smells. Two new metrics (PuF and EncF) were elaborated by the authors.

Sjoberg et al. [178] investigated whether software metrics were the true identifiers for the maintainability of the system or not. Four systems which were practically similar were tested for five groups of bad smells (Feature Envy and God Class). Ferreira et al. [172] revealed that most of the metrics were neglected to use by the developers due to lack of their values. The proposed work generated the threshold values of OO Metrics like LCOM, DIT, coupling factor, afferent couplings etc. Al Dallal [171] performed a study on the impact of methods on the cohesion using metrics. More than 15 metrics were studied for the context of identifying cohesion. Classes that needed refactoring and faulty in nature were revealed. The included-excluded criteria applied upon methods determined the faulty classes based on metric values. Al Dallal [144] introduced a model for OO Metrics to disclose the subclass refactoring possibilities. Openbravo, JHotDraw, GanttProject and JabRef were considered for revealing the internal quality characteristics with reference to metrics. Lozano et al. [140] explored the inter relation between the code smells. The authors considered four types of code smells and proved a strong and mild relationship between the different categories of smells. Three open source projects with their more than 90 releases were investigated for smells with two types of metrics. Saraiva et al. [131] proposed an in-depth investigation of object oriented software maintainability using different kinds of metrics. A number of suggestions were taken from the experts from different countries for categorizing the metrics. Seventeen classifications of metrics were discussed with their impact of maintainability. Zhao et al. [89] explored that the fault tendency in a system was often influenced by the package level metrics which provided an extension of traditional metrics set. Six open source systems were investigated for the metrics developed by Sarkar, Kak, and Rama. Antoniol et al. [88] revealed the conservative approach to make the use of metrics to identify the patterns of design in an OO Environment. A design pattern GUI was developed to specify the patterns, classes and metrics. Higo et al. [78] proposed a method for refactoring effect estimation. The method measured the CK metrics from the original program. The comparison result represents how the complexity of the program was changed by performing the refactoring. A tool was developed based on the proposed method. Elish [72] proposed the classification of refactoring methods based on the measurable effect on (i) internal quality metrics and (ii) software testing effort which is one of the external software quality attribute. It showed a positive correlation between the testing effort and RFC, WMC, NOM, and LOC metrics.

Narendar Reddy and Ananda Rao [58] proposed three experimental cases which successfully indicated for the presence of defects and improvement in the quality of designs after refactoring. These metrics acted as quality indicators with respect to designs considered under ripple effects before and after refactoring. Briand and Briand [38] discussed that design metrics helped in finding the classes containing faults. Coupling, Inheritance, Cohesion measures were focused for identifying their affiliation with the faulty classes. Precise model could be easily developed. Walter and Martenka [36] justified that presence of smells in the software leads to the assembling of patterns, which started affecting software negatively. The patterns related to Large Class Code smells were justified. Two open source software Eclipse Checkstyle and Metrics were validated using the CK Design Metrics.

Table 5 will conclude the results of all the above mentioned approaches.

5. Results and discussions

The results of the systematic literature survey are arranged according to the research questions mentioned in Table 3. But the present literature survey from a total of 238 papers, 38% of them are found on refactoring which are published in the premier journal and in top conferences and workshops (depicted in Fig. 4). Furthermore, 24% of the papers are found on code smells with relevance to performing refactoring. Similarly 13% of the papers are found on anti-patterns in software systems and so on.

Papers on refactoring, code smell and anti-pattern detection are published in numerous journals like the Journals of System and Software, IEEE Transactions on Software Engineering. However conference publications are considered like conferences conducted by IEEE, ACM. We found that 6.98% of research items are published in Springer, 12.41% are published in ACM and a total of 14.34% are published by Elsevier and IEEE is contributing 17.44%. Maximum numbers of papers are accepted in the conference proceedings i.e. 33.33% shown in Fig. 5.

The work performed in this paper is an extension of Al Dallal [2] previous work performed on refactoring in Object Oriented System. As the dataset evaluated is much larger in size than Jehad's one. Secondly, there were few research questions that still left open ended in Jehad's work. Like the tools used for the detection of code smells and the type of code smell identified by specific researcher. The current Systematic literature survey tries to answer some of the demanded issues. The major extensions of the current study are mentioned in Table 6 given below.

5.1. RQ1. What is the current status of refactoring with respect to code smells and anti-patterns?

It has been revealed for the vast literature survey that refactoring is the process of improving the quality of existing software product without changing the external layout. For the realization of this concept, more than 235 papers are inspected. Different authors

Table 5
Approaches used by the authors for performing refactoring.

S No.	Approach	List of citations	#	%
1	Traditional Detection Approach	[23–27,29–32]	9	3.78%
2	Visualization Detection Approach	[33–35,37,42,43]	6	2.52%
3	Semi Automatic Detection Approach	[39,44–46,48–50,52,54–57,59–71,74–77,81–84,86,87,90,91,95,99,106,109,126,130,152,153,161,162,174,181,187]	50	21.009%
4	Metric Based Detection Approach	[28,85,107,155,194,206,233–238,239,240,221,207,203,192,184,178,172,171,144,140,131,89,88,78,72,58,38,36]	32	13.446%
5	Automatic Detection Approach	[10,15,18,20,40,41,47,51,73,80,92–94,96,97,101–105,108,110–113,115,118–125,127–129,132–139,141,142,145,146–151,154,156,158–160,163–170]	67	28.152%
6	Empirical Based Technique	[2,4,5,17,19,53,79,98,100,114,116,117,143,157,173,175–177,179,180,182,183,185,186,188–191,193,195–202,204,205,208–220,83,222,223–232]	64	26.891%

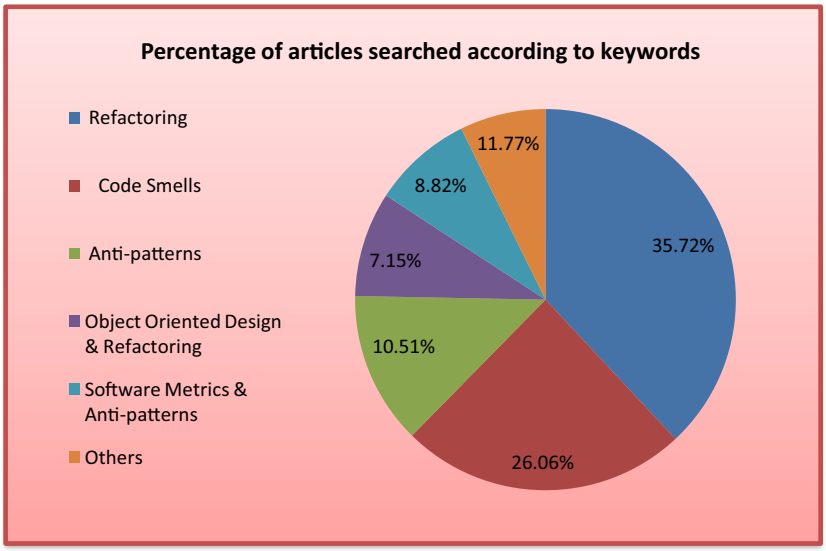

Figure 4. Percentage of papers found according to keywords.

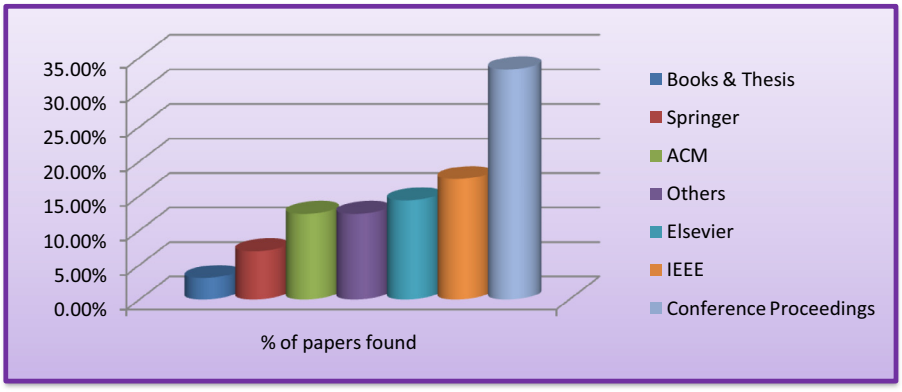

Figure 5. DataBases searched for the systematic literature survey.

applied different criteria for performing refactoring in order to remove the code smells and anti-patterns. More than five detection approaches are discussed in the earlier section of the paper.

The term Refactoring was initially introduced by Opdyke [9] in 1992. However, Webster [12] wrote a book on downside of object oriented design in 1995. In the year 1999 Fowler [1] was the next one to explore the term code smells in detail. Very few studies have been found under this work from the time period of 1990-2000 as shown in Fig. 2. From the year 2001 onwards a noticeable work has been emerged out in the field of refactoring with respect to code smells and anti-patterns detection. However the techniques for the implementation of work vary from traditional to visualization based, semi-automatic, automatic, empirical studies and metric based approaches. The datasets used by the authors

Table 6
Comparison of current work with Jehad's work.

S No.	Issues in current systematic literature survey	Issues managed in Jehad Al Dallal [2] paper
1.	The study is concentrated on 238 research items which includes articles, research publications, thesis, technical reports, books and conference proceedings from leading journals	In Jehad Al Dallal work only 47 primary studies from leading databases
2.	The work had been done since the end of year 2015	The investigation had been done since 2013
3.	Research parameters evaluated by the current work: (a) What is the current status of refactoring with respect to code smells and anti- patterns? (b) What are the different approaches used for the detection of code smells and how the smells are removed using these approaches? (c) What are the different datasets mentioned in the papers for detecting code smells? (d) What are the different tools used by the researchers to identify code smells and performing refactoring? (e) What are the different types of code smells spotted in the papers?	Only few research questions have been answered like: (a) What are the different refactoring activities? (b) What are the different types of approaches used for performing refactoring? (c) What are the different data sets used for performing refactoring?

are generally open source products. Very few authors have worked on the closed source projects. The progress bars in Fig. 6 portray the results. During the year 2013 a tremendous growth has been seen in the refactoring process as compare to code smells and anti-pattern detection.

For the year 2015, a comparative analysis is shown in Fig. 7 for published research items in contrast to the keywords used in the literature survey. i.e. 10 research items are published on refactoring, 5 on anti-patterns, 6 on code smell evaluation and 7 on software maintenance and code smells.

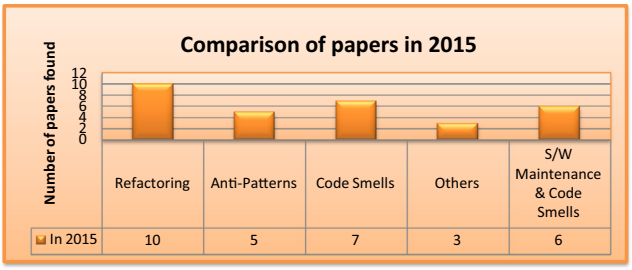

Figure 7. A comparative view with respect to refactoring and code smell detection.

5.2. RQ2. What are the different approaches used for the detection of code smells and how the smells are removed using these approaches?

Different approaches are revealed during the systematic literature survey. From an array of 238 papers, different categories have been formed. Initially manual detection strategies came into limelight which was time consuming in nature. A set of nine papers have been found under this category- [23–27,29–32] which made only a contribution of 3.78% in the literature survey. After that Visualization approaches were originated like Domain Specific Language, Visualized design defect Detection Strategy. A total of 6 papers were instituted – [33–35,37,42,43] which became 2.52% of the entire work. Similarly semi automatic and automatic approaches put in their 21.009% and 28.152% respectively (Citations mentioned in Table 5). It has been noticed that after the automatic detection approaches, it was empirical detection approach which contributed 26.891% in the systematic literature survey. A large number of researchers are performing the studies empirically. Such studies are performed in the form of surveys, systematic studies. The papers belonging to detection strategy like metric based, automatic, semi automatic could be considered under this unit. Remarkable work has been done in the field of metric based detection approaches as metrics are the key aspect for measuring the internal quality of the system. It seems to be the most easiest and interesting approach for the detection of smells. A total of 13.446% of researchers tried on same concept- [28,85,107,155,194,206,233–240,221,207,203,192,184,178,172,171,144,140,131,89,88,78,72,58,38,36]. The datasets and tools used for the removal of code smells are mentioned in the next sections of the systematic literature survey. Fig. 8 given below depicted the results stated above.

5.3. RQ3. What are the different tools used by the researchers to identify code smells and performing refactoring?

In order to reveal code smells different tools are available which disclose smells in the code. Such tools could be automatic in nature i.e. the tool will have a strong GUI which automatically highlights the areas for refactoring and discloses the code smells present in the code e.g. JDeodorant which is an eclipse plug-in is a strong tool for automatic detection of code smells and performs refactoring. But at present the tool is primarily focusing on only four types of smells [51,37,161]. Other tools are also available like PTIDEJ which inputs the source code and generated the equivalent UML Class

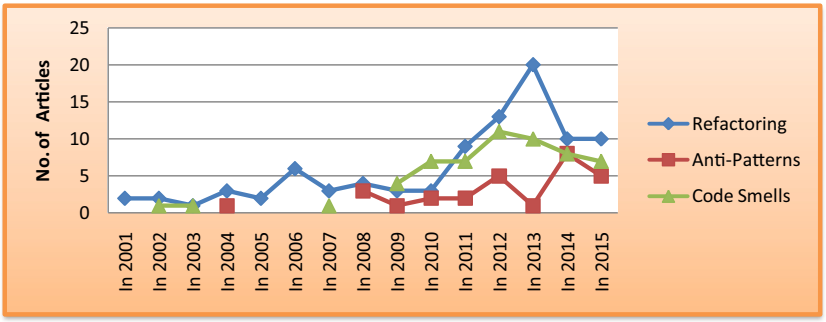

Figure 6. Refactoring with respect to Code Smells and Anti-patterns.

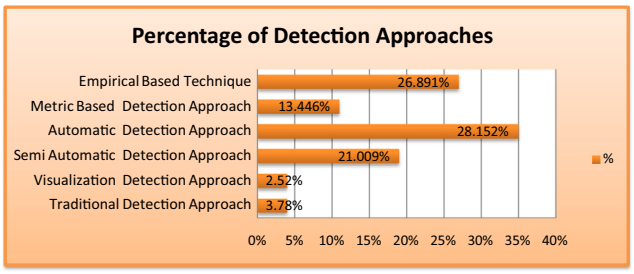

Figure 8. Percentage of approaches used for performing Refactoring.

diagrams followed by the instructions to disclose the code smells and metrics values.

The above mentioned research issue is not managed in the work of Al Dallal [2] paper. Different tools have been used by the researchers using different approaches for revealing the smells which are mentioned in Table 7.

5.4. RQ4. What are the different datasets mentioned in the papers for detecting code smells?

The study revealed multiple datasets which are used by the researchers for the detection of code smells and anti-patterns. Mostly open source projects are taken into consideration. Xerces, JFreeChart, ArgoUML, GanttProject and Eclipse are among the most commonly used datasets for uncovering the bad code smells. However many additional datasets are considered by the researchers like Jedit, Azureus and Log4j. Mozilla Firefox is an open source project but only four studies [107,155,202,225] have worked on that dataset. Multiple versions of the same dataset are validated by different researchers. It has been detected that empirical studies are concentrated highly for the detection of smells using the datasets mentioned. Table 8 reflected the results obtained from the literature survey.

5.5. RQ5. What are the different types of code smells spotted in the papers?

The systematic literature survey aimed at disclosing the work done by the researchers for unmasking the code smells. God Class and Blob are among the most commonly identified code smell whereas Refused Parent Bequest is the one of the least identified code smell (According to the literature survey performed). Table 9 presents the citations of work and related smells.

6. Open issues

The study performed the systematic literature survey of 238 items selected from a set of 325 papers. The target is to identify the applicability of refactoring for the detection of code smells and anti-patterns in object oriented systems. The survey performed is different from Zhang et al. [4] with a focus on aspects like refactoring with respect to code smells and anti-patterns, object oriented design and software maintenance alike to refactoring done by the earlier. Moreover the survey performed by Zang was nearly five years back. However, In Mens and Toure [5] paper the refactoring activity is primarily focused whereas Al Dallal [2] performed the systematic literature survey on exploring Object Oriented Systems for performing "refactoring" only.

The work carried out aimed at different approaches for the detection of code smells on different datasets with different detection tools. We have noticed that future recommendations are still mentioned by the authors in their papers as it is still an open ended research topic. A large number of authors are suggesting the applicability of activity on the large data set with the same technique. Similar datasets are considered by different researchers while they only change the technique for performing refactoring. Few open issues are mentioned below:

(1) The detection of code smells in cross company projects i.e. the datasets from different programming languages.
(2) Another future direction is to consider additional open source systems and industrial projects rather than the datasets mentioned in the work.
(3) The approach is varied in the articles for revealing code smells hence applying different techniques is also a new future direction. However, every approach for revealing code smells has its own weaknesses that should be taken care of.

7. Implications of research & practice

The work carried out acts as the basic platform for researchers who need to be aware of fresh idea in the field of software refactoring for code smell detection. Also, for practitioners who work in the software companies and needed to find the implications of code smell detection. The survey performed will act as a benchmark where different possibilities about refactoring are discussed according to approaches. The area of performing refactoring for code smell detection is an emerging field for the software developers. The detection of code smells reduces the cost of maintenance if the faults are found at the early stages of software development. Such aspect of research will ensure the optimal product output to the users as the software will be extended whenever required. The applicability of the code smell detection tools vary from situation to situation like software quality management, maintenance with respect to code smells detection, improving quality of OO systems and finding faults via refactoring. Thus, the area seems to be the same but have applications in different a facet of software development.

The proposed work will be beneficial to the software industry in improving the quality of the software system by predicting the faulty classes after revealing the code smells. It helps in providing more reliability during maintenance phases by predicting code smells, anti-patterns and faults before delivery of the product. The results produced will be of interests to software engineers, as they gain knowledge about the current scenario of work in the field of code smell detection during refactoring. The study will be valuable for software engineers and managers for improving their maintenance activities by eliciting the code smells.

8. Threats to validity

The basic threat to validity is the application of term refactoring with respect to code smell detection. As software quality assurance is a gigantic area and finding out the research papers on performing refactoring for the detection of smells is a tedious job. However we remain very careful while the selection of research papers and articles. Similarly the word "patterns" is multidisciplinary in nature so while searching papers from different branches like medical, food processing, material sciences, biomechanics, nano-technology is also found. Such papers are discarded from the systematic survey. The data is extracted from the sources mentioned in Table 4 by single author and manually validated by the second author. The quality evaluation of the work is performed by independent reviewer in the field of software engineering. In case of disagreement the solution is made by comparing the results again and correcting them. For keeping the survey fair the papers are selected according to the inclusion – exclusion criteria.

Table 7
Tools used for refactoring and code smell detection.

Approach	Tool used	Aspects identified by the tools
Visualization Based Detection	Stench Blossom [40]	Feature Envy, Large Class, Long Method, Data Clumps
Machine Based Approach	Weka [56]	Data Class, Feature Envy and God Class
Template Matching Approach	C^3PO [68]	Refactoring LiSP Programs
Empirical Detection Strategy	Columbus Wrapper Framework [202]	Generating object oriented metrics values
	Saat [160]	Produces parameter values for anti-patterns detection
	WebDyn [228]	Performs refactoring
	Pinot [179]	Extracting design patterns
Semi Automatic Detection	Jcosmo [14]	Large class, Feature Envy
	Wrangler [62]	Performs Refactoring
	GenReferee [64]	Performs Refactoring
	Jsart [112]	Push up and Push down Refactoring
	ReuseTool [113]	To reuse the OO framework
	SACSEA [125]	God Class, Feature Envy, Data Class, Message Chain Class Long Parameter List
	DND Refactoring [171]	Performs Refactoring
Automatic Detection Approach	InCode [20,130]	Data Class, Feature Envy and God Class
	Ptidej [92,96]	Reverse engineering tool and detects more than 15 code smells
	LBSDetectors [110]	Produces lexicon bad smells
	ReFIT (Eclipse plug-in) [111]	Acceptance Test Refactoring
	JCodeCanine (Eclipse plug-in) [169]	Duplicated Code, Data Class, Switch Statement and Feature Envy
	WebScent [127]	Reveals embedded code smells
	BSDT [128]	Large Class, Data Class, Lazy Class, Parallel Inheritance Hierarchies
	SafeRefactor [136]	Performs Refactoring
	CamFort [141]	Refactoring of Fortran language
	JDEv [145]	Duplicated Code, Long Method, Large Class, Long Parameter List
	PosDef [148]	Outputs design flaws
	Other Eclipse plug- in [20,51,103,105,108,111,119,121,124,128,130,136,137,139,147,159,163–165,167,233]	Uncovers Some of the smells mentioned by Fowler [1]
	Bespoke [80]	15 types of refactorings
Metric Based Detection	Iplasma [28,133]	God class, Data class, Refused parent bequest, Feature envy etc.
	Crocodile [33]	Four refactorings: Move method, Move attribute, Extract class and Inline class.
	JFactor [232,236]	Extract Method Refactoring
	Maisa [237]	Simple object-oriented metrics, Pattern mining results and Pattern-based metrics
	FindBug [207]	Error Collection in source code
	PMD [188,192]	Identify primary problems in code like Dead Code, etc.
	Jdeodorant [122,132]	God Class, Type Check, Feature Envy, Long Method

9. Conclusion

The work done is the systematic literature survey analyzed the detection techniques for revealing the code smells for refactoring in object oriented systems. From four digital libraries a set of 1053 research items were searched, which were later on refined to 238 articles after applying inclusion-exclusion criteria. The work explored from the systematic literature survey concluded that identification and analysis of code smells is an exceedingly energetic field of research.

The work is an extension of Al Dallal's [2] previous work performed on refactoring in Object Oriented Systems. As the dataset evaluated is much larger in size than Jehad's. Secondly there were some research questions; that were still left open in Jehad's work. Like the tools used for the detection of code smells and the type of code smell identified by specific researcher.

Some of the papers included in the survey are original research papers while few are empirical performance articles. However most of the work done in this field is from academic world i.e. 33.33% articles are from conference proceedings while 3.10% work is mentioned in books and thesis. While some of the articles which concentrated towards empirical investigation are from industry showing that industry is also connected with this field. We have disclosed basically six types of detection approaches- traditional method, visualization based technique, automatic method, semi-automatic method, empirical studies and metric based method.

28.152% of the researchers applied automatic detection approaches for uncovering code smells, while 26.891% of them performed empirical studies. Xerces, JFreeChart and ArgoUML are among the mostly used dataset in the articles. Large numbers of datasets are considered in empirical studies hence providing rich ideas to researchers in this field in near future. Some studies compared the datasets for revealing impact of code smells. Some of the datasets used in the articles are same but the detection strategy varies. God Class and Feature Envy smell is highly revealed by most of the studies. Finally it has been observed that most of the datasets are focusing on open source systems. The work carried out by most of the authors for revealing the code smells is based on same company products only.

Acknowledgements

I would like to thanks IKGPTU Jalandhar for providing me the opportunity to study as a research scholar. We are thankful to

Table 8
Datasets used for code smell detection.

Data set	#	Citations
Xerces	15	[15,32,79,100,102,106,118,126,143,159, 183,195,210,215,218]
JFreeChart	9	[20,104,108,132,143,157,159,210,215]
ArgoUML	10	[79,96,117,118,126,143,156,157,195,210]
GanttProject	8	[32,79,215,144,102,106,115,195]
Eclipse	9	[55,100,106,117,146,156,167,180,219]
Jedit	5	[142,146,147,218,219]
Azureus	4	[118,126,195,219]
Log4j	4	[15,79,195,218]
Apache	4	[15,55,156,159]
Hsqldb	4	[98,142,157,218]
JHotDraw	4	[32,106,215]
FireFox	4	[107,155,202,225]
Others		
Rhino	2	[117]
Quick Uml	2	[32,195]
Tyrant	2	[98,157]
Columba	2	[129,158]
Jflex	2	[104,132]
JGIT	2	[129,158]
GESA, SMOS	2	[147]
Nutch	2	[105,159]
MegaMek, JasperReports, Antlr, PDFBox, Velocity	1	[98]
Ice Hockey Manager	1	[105]
Mylyn	1	[117]
SweetHome	1	[218]

Table 9
Code Smell detected.

Name of smell	Citations
God Class	[15,20,22,27,56,100,101,132,133,180,185,212,19,178]
Blob	[12,36,61,79,106,109,118,126,183,195,206]
Feature Envy	[20,56,90,108,109,132,146,147,165,185,178]
Functional Decomposition	[79,106,118,195,206]
Long Method	[41,90,132,133,145,161]
Shotgun Surgery	[109,180,183,195]
Data Class	[20,22,56,133]
Dulipcated Code	[55,133,145]
Large Class	[90,109,144]
Divergant Change	[36,109,185]
Middleman	[55,142]
Data Clumps	[55,156]
Sphagetti Code	[195,206]
Swiss Army Knife	[116,118]
Refusec Parent Bequest	[130]
Long Parameter List	[145]

the reviewer and editor for the valuable suggestions and guidelines. The ideas provided by them were really worthy for attaining the perfection in the work. We are also grateful to the collegeous and family members for their continuous support.

Appendix A. Quality evaluation form 1

Did the systematic review refer to software Refactoring and Code smells detection? Mark yes or no	o Y o N
The paper included in the systematic literature survey is concerned to applying refactoring with respect to Code smells. However the articles could be a case study, experimental study or research paper	

If Appendix A is answered satisfactorily by the reviewer then moved on to Appendix B.

Appendix B. Quality evaluation form 2

Did the study mention the type of software code smells? Mark yes or no	o Y o N
Did the study categorize the software code smells?	o Y o N
Did the paper aimed at disclosing different code smell detection approaches?	o Y o N
Did the study revealed different types of tools used for code smell detection?	o Y o N
Did the code smells used by different authors is highlighted or not?	o Y o N
Did the study uncovers the data values used in different papers	o Y o N

If Appendix B is answered satisfactorily by the reviewer then moved on to Appendix C.

Appendix C. Quality evaluation form 3

Is there clear statement of the findings?	o Y o N
Was the data reported sufficient for comparative analysis?	o Y o N
Was the subject system size considerable?	o Y o N

Appendix D. Data extraction form

Values	Detailed information
Date of data extraction	Mentioned in Table 4
Bibliographic data	Paper title, Author, year, source of information
Category of database	Mentioned in Appendix E
Study aims/context/ application domain	What are the aims of the study, i.e. search focus, i.e. the research areas the paper focus on
Study design	Classification of study – Code smells Evaluation, Categories of Code smells detection approaches based on visualization and comparative analysis
What do you mean by Code smells detection technique	It refers to uncovering the code smells in the source code, followed by approaches
How was comparison carried out?	Based on the articles every year, tools used by the authors for code smell detection and type of dataset
Subject system	How the data was collected: it refers to the subject system and its size
Data analysis	Data analysis, i.e. category source representation and match Detection technique are extracted
Study findings	Major findings are from primary sources based on the keywords mentioned in Table 4

Appendix E. Information sources with relevant percentage

Sources of information	% of papers found (%)
Books & thesis	3.10
Springer	6.98
ACM	12.41
Others	12.41
Elsevier	14.34
IEEE	17.44
Conference proceedings	33.33

Appendix F. Acronyms

Acronym	Full form
OOS	Object Oriented System
CBO	Coupling Between Objects
LCOM	Lack Of Cohesion Of Methods
DIT	Depth Of Inheritance Tree
WMC	Weighted Methods For Class
DÉCOR	Defect Detection For Correction
UML	Unified Modeling Language
CFD	Control-Flow Diagrams
XML	Extensible Markup Language
GUI	Graphical User Interface
OCL	Object Constraint Language
AST	Abstract Syntax Tree
RINV	Right To Inverse
XMI	Xml Metadata Interchange
C3PO	Code Conversion to Prevent Obsolescence
DFA	Deterministic Finite Automata
DAM	Data Access Metric
DCC	Direct Class Coupling
CAM	Computer Aided Manufacturing
PTIDEJ	Pattern Trace Identification and Detection in Java
CBS	Component Based Software
IDS	Immune based Detection Strategy
BBN	Bayesian Belief Networks
GQM	Goal Question Metric
HIST	Historical Information for Smell detection
CKM	Chidamber and Kemerer Metric
HTN	Hierarchical Task Network
TACO	Textual Analysis for Code smell detection
MIS	Maximal Independent Set
AOP	Aspect-Oriented Programming

References

[1] Fowler Martin, Beck Martin, Brant Kent, Opdyke John, Roberts William. Refactoring-improving the design of existing code. 1st ed. Addison-Wesley; 1999.

[2] Al Dallal Jehad. Identifying refactoring opportunities in object-oriented code: a systematic literature review. Inf Softw Technol 2015;58:231–49.

[3] Misbhauddin Mohammed, Alshayeb Mohammad. UML model refactoring: a systematic literature review. Empirical Softw Eng 2015;20(1):206–51. Springer.

[4] Zhang Min, Hall Tracy, Baddoo Nathan. Code bad smells: a review of current knowledge. J Softw Maint Evol Res Pract 2011;23:179–202.

[5] Mens Tom, Tourwe Tom. A survey of software refactoring. IEEE Trans Softw Eng 2004;30(2).

[6] Kitchenham Barbara, Pretorius Rialette, Budgen David, Pearl Brereton O, Turner Mark, Niazi Mahmood, Linkman Stephen. Systematic literature reviews in software engineering: a tertiary study. Inf Softw Technol 2010;52:792–805.

[7] Brereton Pearl, Kitchenham Barbara A, Budgen David, Turner Mark, Khalil Mohamed. Lessons from applying the systematic literature review process within the software engineering domain. J Syst Softw 2007;80:571–83.

[8] Stapic Zlatko, Lopez Eva Garcia, de Marcos Ortega Antonio García Cabot Luis, Strahonja Vjeran. Performing systematic literature review in software engineering. In: Central European conference on information and intelligent systems; 2012. p. 441–93.

[9] Opdyke William F. Refactoring object oriented frameworks Ph. D Thesis. University of Illinois at Urbana-Champaign Illinois; 1992.

[10] Astels Dave. Refactoring with UML. In: Proc. 3rd international conference on extreme programming and flexible processes in software engineering, Italy; 2002. p. 67–70.

[11] Brown WJ, Malveau RC, Brown WH, McCormick III HW, Mowbray TJ. Anti patterns: refactoring software, architectures, and projects in crisis. 1st ed. New York: Wiley; 1998.

[12] Webster BF. Pitfalls of object oriented development. 1st ed. M & T Books; 1995.

[13] Mantyla Mika. Bad smells in software – a taxonomy and an empirical study Ph. D Thesis. Helsinki University of Technology; 2003.

[14] van Emden Eva, Moonen Leon. Java quality assurance by detecting code smells. In: Proc. of ninth working conference on reverse engineering. IEEE; 2002. p. 97–106.

[15] Olbrich Steffen M, Cruzes Daniela S, Sjberg Dag IK. Are all code smells harmful? In: A study of god classes and brain classes in the evolution of three open source systems, 26th IEEE conference on software maintenance 2010 IEEE.

[16] van Emden Eva, Moonen Leon. Assuring software quality by code smell detection. In: Proceedings of the 9th Working Conference on Reverse Engineering (WCRE). IEEE; 2002. p. 97–106.

[17] 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. In: IEEE/ACM 37th IEEE International Conference on Software Engineering. IEEE; 2015.

[18] Liu Hui, Guo Xue, Shao Weizhong. Monitor-based instant software refactoring. IEEE Trans Software Eng 2013;39(8):1112–26.

[19] Santos, MMendonça Jose Amancio, Cleber Manoel Gomes De Pereira. The problem of conceptualization in god class detection: agreement, strategies and decision drivers. J Softw Eng Res Dev 2014.

[20] Ganea George, Verebi Ioana, Marinescu Radu. Continuous quality assessment with in code. Sci Comput Program 1;1015:1–15.

[21] Yamashita Aiko, Moonen Leon. Do code smells reflect important maintainability aspects? IEEE Int Conf Softw Maint, ICSM; 2012.

[22] Fontana Francesca Arcelli, Ferme Vincenzo, Zanoni Marco, Yamashita Aiko. Automatic metric thresholds derivation for code smell detection. In: 2015 IEEE/ACM 6th International Workshop on Emerging Trends in Software Metrics. p. 44–53.

[23] Allen R, Garlan D. A formal basis for architectural connection. ACM Trans Softw Eng Methodol 1997;6(3):213–49.

[24] Garlan D, Allen R, Ockerbloom J. Architectural mismatch: why reuse is so hard. IEEE Softw 1995;12(6):17–26.

[25] Noble James. Classifying relationship between object-oriented design patterns. In: Proceedings software engineering conference IEEE; 1998. p. 98–107.

[26] Travassos G, Shull F, Fredericks Michael, Basil Victor R. Detecting defects in object-oriented designs: using reading techniques to increase software quality. In: Proc. 14th conf. object-oriented programming, systems, languages, and applications; 1999. p. 47–56.

[27] Smith Connie U, William Lloyd G. Software performance anti-patterns. ACM Soft Eng Res 2000:127–36.

[28] Marinescu R. Detection strategies: metrics-based rules for detecting design flaws. In: Proc. 20th int conf software maintenance; 2004. p. 350–9.

[29] Dashofy EM, vander Hoek A, Taylor RN. A comprehensive approach for the development of modular software architecture description languages. ACM Trans Software Eng Methodol 2005;14(2):199–245.

[30] Munro MJ. Product metrics for automatic identification of "bad smell" design problems in java source-code. In: Proc. 11th IEEE int software metrics sympos; September 2005.

[31] Alikacem EH, Sahraoui H. Generic metric extraction framework. In: Proc. 16th Int. Workshop Software Measurement and Metrik Kongress; 2006. p. 383–90.

[32] Ghannem Adnane, Boussaidi Ghizlane El, Kessentini Marouane. On the use of design defect examples to detect model refactoring opportunities. Softw Qual J 2015. Springer.

[33] Simon F, Steinbruckner F, Lewerentz C. Metrics based refactoring. In: Proc. fifth european conf. software maintenance and re-eng; 2001. p. 30.

[34] Baudry Benoit, Traon Yves Le, Sunye Gerson, Jezequel Jean-Marc. Measuring and improving design patterns testability. In: Proceedings of the ninth international software metrics symposium (METRICS'03); 2003. IEEE.

[35] Langelier G, Sahraoui HA, Poulin Pierre. Visualization-based analysis of quality for large-scale software systems. In: ACM Inter Conf on automated soft Eng; November 2005. p. 214–3.

[36] Walter Bartosz, Martenka Pawel. Looking for patterns in code bad smells relations. In: 2011 fourth international conference on software testing, verification and validation workshops, IEEE; 2011. p. 465–6.

[37] Gold David Binkley, Harman N, Mahdavi M Zheng Li, Wegener K. Dependence anti patterns. In: 23rd IEEE/ACM international conference on automated software engineering – workshops, 2008. ASE workshops 2008, 23rd IEEE/ACM international conference on.

[38] Briand Lionel C. A comprehensive empirical validation of design measures for object oriented systems. In: Fifth international conference on software metrics symposium; 1998, IEEE. p. 246–57.

[39] Moha Naouel, Gueheneuc Yann-Gael, Duchien Laurence, Meur Anne-Francoise Le. DECOR: a method for the specification and detection of code and design smells. IEEE Trans Softw Eng 2010;36(1):20–36.

[40] Murphy-Hill Emerson, Black Andrew P. An interactive ambient visualization for code smells. SOFTVIS'10, October 25–26; 2010 [ACM].

[41] Bryton Sergio, Abreu Fernando Brito e, Monteiro Miguel. Reducing subjectivity in code smells detection: experimenting with the long method. In: 2010 seventh international conference on the quality of information and communications technology. IEEE; 2010.

[42] Tekin Umut, Erdemir Ural, Buzluca Feza. Mining object-oriented design models for detecting identical design structures. IEEE 43 IWSC 2012; 2012. p. 43–9.

[43] Soodeh Hosseini, Abdollahi Azgomi Mohammad. UML model refactoring with emphasis on behavior preservation. In: 2nd IFIP/IEEE international symposium on theoretical aspects of software engineering; 2008. p. 125–8.

[44] Noble James. Towards a pattern language for object oriented design. IEEE; 1998.

[45] Tie Feng, Zhang Jiachen, Wang Hongyuan, Wang Xian. Software design improvement through anti-patterns identification. In: Proceedings of the 20th IEEE international conference on software maintenance (ICSM'04); 2004 IEEE.

[46] Baudry Benoit, Le Traon Yves, Cedex Rennes, Sunye Gerson. Improving the testability of UML class diagrams. In: First international workshop on testability assessment; 2004. p. 70–80 [IEEE].

[47] Ballis D, Baruzzo A, Comini M. A Minimalist visual notation for design patterns and antipatterns. In: Fifth international conference on information technology: new generations. IEEE; 2008.

[48] Correa Alexandre, Werner Claudia. Refactoring object constraint language specifications. Softw Syst Model 2007;6:113–38.

[49] Van Rompaey Bart, Demeyer Serge, Rieger Matthias. On the detection of test smells: a metrics-based approach for general fixture and eager test. IEEE Trans Software Eng 2007;33(12):800–17.

[50] Murphy-Hill Emerson, Black Andrew P, Dig Danny, Parnin Chris. Gathering refactoring data: a comparison of four methods, WRT'08, October 19; 2008 [ACM].

[51] Kempf Martin, Kleeb Reto, Klenk Michael, Sommerlad Peter. Cross language refactoring for eclipse plug-ins. ACM; 2008.

[52] Nguyen Trung, Pooley Rob. Effective recognition of patterns in object-oriented designs. In: Fourth international conference on software engineering advances; 2009. p. 320–5.

[53] Abebe Mesfin, Chonbuk Cheol-Jung Yoo. Trends, opportunities and challenges of software refactoring: a systematic literature review. Int J Softw Eng Its Appl 2014;8(6):299–318.

[54] Cornelio Marcio, Cavalcanti Ana, Sampaio Augusto. Sound refactorings, science of computer programming. Sci Comput Program 2010;75(3):106–33.

[55] Zhang Min, Baddoo Nathan, Hall Tracy. Prioritizing refactoring using code bad smells. In: 2011 Fourth international conference on software testing, verification and validation workshops IEEE.

[56] Fontana Francesca Arcelli, Zanoni Marco. On investigating code smells correlations. In: Proc. – 4th IEEE international conference on software testing, verification, and validation workshops, ICSTW; 2011. p. 474–5.

[57] Shuai Jiang, Huaxin Mu. Design patterns in object oriented analysis and design. 978-1-4244-9698-3/11/$26.00; 2011 [IEEE].

[58] Narendar Reddy K, Ananda Rao A. A quantitative evaluation of software quality enhancement by refactoring using dependency oriented complexity metrics. In: Second international conference on emerging trends in engineering and technology, ICETET-09; 2009. p. 1011–8 [IEEE].

[59] Yamashita Aiko, Moonen Leon. Exploring the impact of inter-smell relations on software maintainability: an empirical study. In: Proceedings – international conference on software engineering. IEEE; 2013. p. 682–91.

[60] Liu Hui, Ma Zhiyi, Shao Weizhong, Niu Zhendong. Schedule of bad smell detection and resolution: a new way to save effort. IEEE Trans Softw Eng 2012;38(1).

[61] Cortellessa Vittorio, Marco Antinisca Di, Trubiani Catia. An approach for modeling and detecting software performance antipatterns based on first-order logics. Softw Syst Model 2014;13(1):391–432.

[62] Li Huiqing, Thompson Simon. Let's make refactoring tools user-extensible, WRT, June 01 2012, Rapperswil, Switzerland; 2012 [ACM].

[63] Murphy-Hill Emerson, Black Andrew P. Programmer-friendly refactoring errors. IEEE Trans Softw Eng 2012;38(6):1417–31.

[64] Liu Hui, Niu Zhendong, Man Zhiyi, Shao Weizhong. Identification of generalization refactoring opportunities. Autom Softw Eng 2013;20(1):81–110.

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

[66] Mahmoud Anas, Niu Nan. Supporting requirements traceability through refactoring. In: 2013 21st IEEE international requirements engineering conference, RE 2013 – Proceedings, Requirements Eng 2014;19:309–329.

[67] Kessentini Marouane, Mahaouachi Rim, Ghedira Khaled. What you like in design use to correct bad-smells? Softw Qual J 2013;21:551–71.

[68] Leitao Antonio Menezes. A formal pattern language for refactoring of lisp programs. In: Proceedings of the sixth European conference on software maintenance and reengineering (CSMR'02) 1534–5351/02; 2002 [IEEE].

[69] Hegedus Peter. Revealing the effect of coding practices on software maintainability. In: 2013 IEEE international conference on software maintenance.

[70] Wang Meng, Gibbons Jeremy, Matsuda Kazutaka, Zhenjiang Hu. Refactoring pattern matching. Sci Comput Program 2013;78:2216–42.

[71] Ujhelyi Zoltan, Horvath Akos, Varro Daniel. Anti-pattern detection with model queries: a comparison of approaches; 2014. p. 293–302 [IEEE].

[72] Elish Karim O, Alshayeb Mohammad. Investigating the effect of refactoring on software testing effort. In: 16th Asia-Pacific software engineering conference. IEEE; 2009. p. 29–34.

[73] Vidal Santiago A, Marcos Claudia, Andres Diiz-Pace J. An approach to prioritize code smells for refactoring. Automated Softw Eng 2014.

[74] Yamashita Aiko. Assessing the capability of code smells to support software maintainability assessments: empirical inquiry and methodological approach Ph. D Thesis. University of Oslo; 2012.

[75] Kessentini Wael, Kessentini Marouane, Sahraoui Houari, Bechikh Slim, Ouni Ali. Cooperative parallel search-based software engineering approach for code-smells detection. IEEE Trans Software Eng 2014;40(9).

[76] Khan Yasser A, El-Attar Mohamed. Using model transformation to refactor use case models based on antipatterns. Inf Syst Front. http://dx.doi.org/10.1007/s10796-014-9528-z [Springer].

[77] Unterholzner Martin. Improving refactoring tools in Smalltalk using static type inference. Sci Comput Program 2014;96:70–83.

[78] Higo Yoshiki, Matsumoto Yoshihiro, Kusumoto Shinji, Inoue Katsuro. Refactoring effect estimation based on complexity metrics. In: 19th Australian conference on software engineering; 2008. p. 219–28 [IEEE].

[79] Ghannem Adnane, El Boussaidi Ghizlane, Kessentini Marouane. On the use of design defect examples to detect model refactoring opportunities. Software Qual J. http://dx.doi.org/10.1007/s11219-015-9271-9.

[80] Gatrell M, Counsell S. Science of computer programming 102 (2015) 44–56. The effect of refactoring on change and fault-proneness in commercial C # software.

[81] Tilevich Eli, Smaragdakis Yannis. Binary refactoring: improving code behind the scenes; 2005. ICSE'05 ACM.

[82] Wohlfarth Sven, Riebisch Matthias. Evaluating alternatives for architecture-oriented refactoring. In: Proceedings of the 13th annual IEEE international symposium and workshop on engineering of computer based systems (ECBS'06). IEEE; 2006.

[83] Mealy Erica, Strooper Paul. Evaluating software refactoring tool support. In: Proceedings of the 2006 Australian software engineering conference (ASWEC'06). IEEE; 2006.

[84] Piveta Eduardo, Araujo Joao, Pimenta Marcelo, Moreira Ana, Guerreiro Pedro, Price R Tom. Searching for opportunities of refactoring sequences: reducing the search space; 2008. p 319–26 [IEEE].

[85] Alshayeb Mohammad. Refactoring effect on cohesion metrics. In: International conference on computing, engineering and information. IEEE; 2009. p. 3–7.

[86] Meananeatra Panita, Rongviriyapanish Songsakdi. Using software metrics to select refactoring for long method bad smell. Comput Inform Technol Softw Eng 2011:492–5.

[87] Abdel-hamid, Noaman Amr. Refactoring as a lifeline: lessons learned from refactoring. IEEE; 2013.

[88] Antoniol G, Fiutem R, Cristoforteii L. Using metris to identify design patterns in object oriented software. In: Fifth international conference on software metrics symposia. IEEE; 1998. p. 22–34.

[89] Zhao Yangyang, Yang Yibiao, Hongmin Lu, Zhou Yuming, Song Qinbao, Xua Baowen. An empirical analysis of package-modularization metrics: Implications for software fault-proneness. Inf Softw Technol 2015;57:186–203.

[90] Wongpiang Ratapong, Muenchaisri Pornsiri. Selecting sequence of refactoring techniques usage for code changing using greedy algorithm. IEEE; 2013.

[91] Chu Peng-Hua, Hsueh Nien-Lin, Chen Hong-Hsiang, Liu Chien-Hung. A test case refactoring approach for pattern-based software development. Softw Qual J 2012;20(1):43–75.

[92] Gueheneuc Yann Gael, Albin-Amiot Herve, de Nantes Ecole des Mines. Using design patterns and constraints to automate the detection and correction of inter-class design defects. In: Paper accepted at TOOLS USA; 2001.

[93] Tokuda Lance, Batory Don. Evolving object-oriented designs with refactorings automated software engineering. Kluwer Academic Publishers. Manufactured in The Netherlands; 2001, vol. 8. p. 89–120.

[94] Ekman Torbjorn, Schafer Max, Verbaere Mathieu. Refactoring is not (yet) about transformation. In: Proc. of 2nd workshop on refactoring tools, ACM; 2008.

[95] Bansiya Jagdish, Davis Carl G. A hierarchical model for object oriented design quality assessment. IEEE Trans Software Eng 2002;28(1):4–17.

[96] Gueheneuc Yann Gael. A systematic study of UML class diagram constituents for their abstract and precise recovery. In: 11th Asia-Pacific conference on soft engg; November–December 2004. p. 265–74.

[97] Washizaki Hironori, Fukazawa Yoshiaki. A technique for automatic component extraction from object-oriented programs by refactoring. Sci Comput Program 2005;56:99–116.

[98] Counsell S, Hassoun Y, Common refactorings, a dependency graph and some code smells: an empirical study of Java OSS. IEEE international symposium on empirical soft engg.; 2006. p. 288–96.
[99] Gueheneuc Yann Gael, Antoniol Giuliano. DeMIMA: a multilayered approach for design pattern identification. IEEE Trans Softw Eng 2008;34(5):667–84.
[100] Vaucher Stephane, Khomh Foutse, Moha Naouel, Gueheneuc Yann-Gael. Tracking design smells: lessons from a study of god classes. In: 16th working conference on reverse engg.; 2009.
[101] Llano Maria Teresa, Pooley Rob. 2009 fourth international conference on software engineering advances, UML specification and correction of object-oriented anti-patterns; 2009 [IEEE].
[102] Hassaine Salima, Khomh Foutse, Gueheneuc Yann-Gael, Hamel Sylvie. IDS: an immune-inspired approach for the detection of software design smells. In: 7th IEEE inter conference on the quality of infor and comm tech; September–October 2010. p. 343–8.
[103] Fontana Francesca Arcelli, Zanoni Marco. A tool for design pattern detection and software architecture reconstruction. Inf Sci 2011;181(7):1306–24.
[104] Chatzigeorgiou Alexander, Manakos Anastasios. Investigating the evolution of bad smells in object-oriented code. In: Proceedings – 7th international conference on the quality of information and communications technology, QUATIC. IEEE; 2010. p. 106–15.
[105] Tsantalis Nikolaos, Chatzigeorgiou Alexander. Identification of refactoring opportunities introducing polymorphism. J Syst Softw 2010;83:391–404.
[106] Khomh Foutse, Vaucher Stephane, Guéhéneuc Yann Gael, Sahraoui Houari. BDTEX: A GQM-based Bayesian approach for the detection of anti-patterns. J Syst Softw 2011;84(4):559–72.
[107] Singh Satwinder, Kahlon KS. Effectiveness of refactoring metrics model to identify smells and error prone classes in open source software. ACM SIGSOFT Soft Engg Notes 2011;36(5):1–11.
[108] Fokaefs Marios, Tsantalis Nikolaos, Stroulia Eleni, Chatzigeorgiou Alexander. Identification and application of Extract Class refactorings in object-oriented systems. J Syst Softw 2012;85(10):2241–60.
[109] Palomba Fabio, Bavota Gabriele, Di Penta Massimiliano, Oliveto Rocco, Poshyvanyk Denys. Mining version histories for detecting code smells. IEEE Trans Softw Eng 2015;41(5):462–89.
[110] Abebe Surafel Lemma, Haiduc Sonia, Tonella Paolo, Marcus Andrian. The effect of lexicon bad smells on concept location in source code. In: 2011 IEEE 11th international working conference on source code analysis and manipulation; 2011. p 125–34.
[111] Borg Rodrick, Kropp Martin. Automated acceptance test refactoring, WRT'11, May 22, 2011, Waikiki, Honolulu, HI, USA Copyright; 2011. p. 15–21 [ACM].
[112] Maruyama Katsuhisa, Omori Takayuki. A security-aware refactoring tool for java programs. WRT'11, May 22, 2011Copyright 2011. p. 22–8.
[113] Oliveira Toacy C, Alencar Paulo, Cowan Don. ReuseTool – an extensible tool support for object-oriented framework reuse. J Syst Softw 2011;84(12):2234–52.
[114] 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 2012;37(2):1–10.
[115] Fontanaa Francesca Arcelli, Braione Pietro, Zanoni Marco. Automatic detection of bad smells in code: an experimental assessment. J Object Technol 2012;11(2):1–38.
[116] Romano Daniele, Raila Paulius. Analyzing the impact of anti-patterns on change-tendency using fine- grained source code changes. In: Proc of the 19th working conference on reverse engineering (WCRE). IEEE Computer Society Press; 2012.
[117] Khomh Foutse, Penta Massimiliano Di, Gueheneuc Yann Gael, Antoniol Giuliano. An exploratory study of the impact of anti-patterns on class change- and fault-tendency. Springer Science Business Media, LLC; 2012.
[118] Maiga Abdou, Ali Nasir, Bhattacharya Neelesh, Sabane Aminata, Gueheneuc Yann-Gael, Aimeur Esma. SMURF: A SVM-based incremental anti-pattern detection approach. In: Presented at 19th working conference on reverse engineering; October 2012. p. 466–75.
[119] Kaur Sharanpreet, Singh Satwinder. Spotting & eliminating type checking code smells using eclipse plug-in: Jdeodorant. Int J Comput Sci Commun Eng 2016;5(1).
[120] Brown Christopher, Hammond Kevin, Danelutto Marco, Kilpatrick Peter. A language-independent parallel refactoring framework; 2012. p. 54–58 [ACM].
[121] Schafer Max. Refactoring tools for dynamic languages, WRT June 01 2012, Rapperswil, Switzerland. Copyright ACM.
[122] Christopoulou Aikaterini, Giakoumakis EA, Zafeiris Vassilis E, Soukara Vasiliki. Automated refactoring to the Strategy design pattern. Inform Soft Technol, Inform Soft Technol 2012;54:1202–14.
[123] Perez Javier, Crespo Yania. Computation of refactoring plans from refactoring strategies using HTN planning, WRT; June 01 2012. p 24–31 [ACM].
[124] Einarsson Hafsteinnpor, Neukirchen Helmut. An approach and tool for synchronous refactoring of UML diagrams and models using model-to-model transformations; 2012. p 16–23 [ACM].
[125] Peters Ralph, Zaidman Andy. Evaluating the lifespan of code smells using software repository mining. In: 2012, 16th European conference on software maintenance and reengineering. IEEE; 2012.
[126] Maiga Abdou, Ali Nasir, Bhattacharya Neelesh, Sabane Aminata, Gueheneuc Yann-Gael, Antoniol Giuliano, Aïmeur Esma. Support vector machines for anti-pattern detection categories and subject descriptors, ASE '12, September 3–7, 2012, Essen, Germany Copyright 12 ACM 2012.
[127] Nguyen Hung Viet, Nguyen Hoan Anh, Nguyen Tung Thanh, Nguyen Anh Tuan, Nguyen Tien N. Detection of embedded code smells in dynamic web applications. In: Proceedings of the 27th IEEE/ACM international conference on automated software engineering – ASE; 2012. p. 282–5.
[128] Danphitsanuphan Phongphan, Suwantada Thanitta. Code smell detecting tool and code smell-structure bug relationship. 2012 Spring Congress on Engineering and Technology IEEE.
[129] Han Ah-Rim, Bae Doo-Hwan. Dynamic profiling-based approach to identifying cost-effective refactorings. Inf Softw Technol 2013;55(6):966–85.
[130] Sjoberg Dag IK, Yamashita Aiko, Anda Bente CD, Mockus Audris, Dyba Tore. Quantifying the effect of code smells on maintenance effort. IEEE Trans Software Eng 2013;39(8):1144–56.
[131] Saraiva Julianade AG, deFrança Micael 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 2015;103:85–101.
[132] Chatzigeorgiou Alexander, Manakos Anastasios. Investigating the evolution of code smells in object-oriented systems. Innovations Syst Softw Eng 2014;10(1):3–18.
[133] Fontana Francesca Arcelli, Ferme Vincenzo, Marino Alessandro, Walter Bartosz, Martenka Pawel. Investigating the impact of code smells on system's quality: an empirical study on systems of different application domains. In: 2013 IEEE international conference on software maintenance. p. 260–69.
[134] Ligu Elvis, Chatzigeorgiou Alexander, Chaikalis Theodore, Ygeionomakis Nikolaos. Identification of refused bequest code smells. IEEE International conference on software maintenance; 2013.
[135] Palomba Fabio, Bavota Gabriele, Penta Massimiliano Di. Detecting bad smells in source code using change history information, 2013 28th IEEE/ACM international conference on automated software engineering, ASE 2013 – proceedings.
[136] Soares Gustavo, Gheyi Rohit, Massoni Tiago. Automated behavioral testing of refactoring engines. IEEE Trans Softw Eng 2013;39(2):147–62.
[137] Arendt Thorsten, Taentzer Gabriele. A tool environment for quality assurance based on the Eclipse Modeling Framework. Automated Softw Eng; 2013, vol. 20, issue 2. p. 141–84 [Springer].
[138] Bavota Gabriele, De Lucia Andrea, Marcus Andrian, Oliveto Rocco. Automating extract class refactoring: an improved method and its evaluation. Empir Softw Eng 2013:1–48.
[139] Kim Tae-Woong, Kim Tae-Gong, Seu Jai-Hyun. Specification and automated detection of code smells using OCL. Int J Softw Eng Its Appl 2013;7(4):35–44. Specification.
[140] Lozano Angela, Mens Kim, Portugal Jawira. Analyzing code evolution to uncover relations between bad smells. IEEE; 2015. p. 2–5.
[141] Orchard Dominic, Rice Andrew. Upgrading FORTRAN Source Code Using Automatic Refactoring, WRT '13; October 27; 2013 [ACM].
[142] Jiau Hewijin Christine, Mar Lee Wei, Chen Jinghong Cox. OBEY: optimal batched refactoring plan execution for class responsibility redistribution. IEEE Trans Software Eng 2013;39(9):1245–63.
[143] Jaafar Fehmi, Yann-Gael, Hamel Sylvie, Khomh Foutse. Analysing anti-patterns static relationships with design patterns. In: Proc. of the first workshop on patterns promotion and anti-patterns prevention, EASST; 2013. vol. 59.
[144] Dallal Jehad Al. Constructing models for predicting extract subclass refactoring opportunities using object-oriented quality metrics. Inf Softw Technol 2012;54.10:1125–41.
[145] Lakshmanan M, Manikandan S. Multi-step automated refactoring for code smell. Int J Res Eng Technol 2014:278–82.
[146] Kumar Swati, Chhabra Jitender Kumar. Two level dynamic approach for feature envy detection. In: 2014 5th international conference on computer and communication technology (ICCCT); 2014. p. 41–6.
[147] Bavota Gabriele, Oliveto Rocco, Gethers Malcom, Poshyvanyk Denys, De Lucia Andrea. Methodbook: recommending move method refactorings via relational topic models. IEEE Trans Software Eng 2014;40(7).
[148] Chaudron Michel RV, Katumba Brian, Ran Xuxin. Automated prioritization of metrics-based design flaws in UML class diagrams 2014 40th Euromicro conference on software engineering and advanced applications IEEE.
[149] Peiris Manjula, Hill James H. Towards detecting software performance anti-patterns using classification techniques. ACM SIGSOFT Software Engineering Notes Page 1–4. Volume 39 Number 1 January 2014.
[150] Jaafar Fehmi, Foutse Khomh, Gu eneuch Yann-Gael, Zulkernine Mohammad. Anti-pattern mutations and fault-proneness. In: 2014 14th International Conference on Quality Software, IEEE.
[151] Mongiovia Melina, Gheyia Rohit, Soaresa Gustavo, Teixeira Leopoldo, Borbab Paulo. Making refactoring safer through impact analysis. Sci Comput Program 2014;93:39–64.
[152] Pandiyavathi, Manochandar. Detection of optimal refactoring plans for resolution of code smells. Int J Adv Res Comput Commun Eng 2014;3(5):6–11.
[153] Palma Francis, Moha Naouel, Guenheneuc Yann Gael. Detection of process anti-patterns: a BPEL perspective. In: 17th IEEE Int. Workshop on Enterprise Distributed Object Computing. September 2013; p. 173–7.
[154] Palma Francis, An Le. Investigating the change-proneness of service patterns and anti-patterns. In: 7th inter. conf. on service-oriented computing and applications IEEE; November 2014. p. 1–8.

[155] Singh Satwinder, Kahlon KS. Object oriented software metrics threshold values at quantitative acceptable risk level. CSI Transactions on ICT, Springer; November 2014, vol. 2, no. 3. p. 191–205.
[156] Hall Tracy, Zhang Min. Some code smells have a significant but small effect on faults. ACM Trans Softw Eng Methodol 2014;24(4):33:1–33:39. ACM.
[157] Dexun Jiang, Peijun Ma, Xiaohong Su, Tiantian Wang. Functional over-related classes bad smell detection and refactoring suggestions. Int J Softw Eng Appl (IJSEA) 2014;5(2):29–47.
[158] Han Ah-Rim, Bae Doo-Hwan, Cha Sungdeok. An efficient approach to identify multiple and independent Move Method refactoring candidates. Inf Softw Technol 2015;59:53–66.
[159] Gaitani Maria Anna G, Zafeiris Vassilis E, Diamantidis NA, Giakoumakis EA. Automated refactoring to the NULL OBJECT design pattern. Inf Softw Technol 2015;59:33–52.
[160] Ammar Boulbaba Ben, Bhiri Mohamed Tahar. Pattern-based model refactoring for the introduction association relationship. J King Saud University – Comput Inform Sci 2015;27(2):170–80.
[161] Palomba Fabio. Textual analysis for code smell detection. IEEE/ACM 37th IEEE international conference on software engineering; 2015. p 769–71.
[162] 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 2015;65:147–65.
[163] Morales Rodrigo. Towards a framework for automatic correction of anti-patterns 2015 IEEE; 2015. p. 603–4.
[164] Liu Hui, Liu Qiurong, Liu Yang, Wang Zhouding. Identifying renaming opportunities by expanding conducted rename refactorings. IEEE Trans Softw Eng 2015;5589.
[165] Nongpong Kwankamol. Integrating "code smells" detection with refactoring tool support. Ph. D Thesis. University of Wisconsin-Milwaukee; 2012.
[166] Mancl Dennis. Refactoring for software migration. IEEE Communications Magazine; October 2001.
[167] Xing Zhenchang, Stroulia Eleni. Refactoring practice: how it is and how it should be supported – an eclipse case study. In: 22nd IEEE international conference on software maintenance (ICSM'06 2006.
[168] Lan Ling, Huang Gang, Wang Weihu, Mei Hong. A middleware-based approach to model refactoring at runtime. In: 14th Asia-Pacific software engineering conference. IEEE; 2007. p. 246–53.
[169] Maruyama Katsuhisa, Tokoda Kensuke. Security-aware refactoring alerting its impact on code vulnerabilities. In: 15th Asia-Pacific software engineering conference; 2008. p. 445–52.
[170] Sondergaard Hans, Thomsen Bent, Ravn Anders P. Refactoring real-time Java profiles. In: 14th IEEE international symposium on object/component/service-oriented real-time distributed computing; 2011 [IEEE].
[171] Al Dallal Jehad. The impact of accounting for special methods in the measurement of object-oriented class cohesion on refactoring and fault prediction activities. J Syst Soft 2012;85(5):1042–57.
[172] Ferreira Kecia AM, Bigonha Mariza AS, Bigonha Roberto S, Mendes Luiz FO, Almeida Heitor C. The journal of systems and software, identifying thresholds for object-oriented software metrics. J Syst Softw 2012;85(2):244–57.
[173] Briand Lionel C, Wust Jurgen, Ikonomovski Stefan V, Lounis Hakim. Investigating quality factors in object-oriented designs: an industrial case study. ICSE '99 Los Angclcs CA ACM; 1999.
[174] Briand Lionel C, Melo Walcelio L, Wust Jurgen. Assessing the applicability of fault-proneness models across object-oriented software projects. IEEE Trans Softw Eng 2002;28(2).
[175] Mantyla Mika, Vanhanen Jari, Lassenius Casper. A taxonomy and an initial empirical study of bad smells in code. In: International conference on software maintenance; 2003. ICSM 2003. Proceedings.
[176] Mens Tom, Demeyer Serge, Bois Bart Du, Stenten Hans, Van Gorp Pieter. Refactoring: current research and future trends. Electron Notes Theor Comput Sci 2003;83(3):483–99.
[177] Fabry Johan, Mens Tom. Language-independent detection of object-oriented design patterns. Comput Lang, Syst Struct 2004;30:21–33.
[178] Sjoberg Dag IK, Anda Bente, Mockus Audris. Questioning software maintenance metrics: a comparative case study, ESEM'12, September 19–20, 2012, Lund, Sweden. Copyright 2012 ACM.
[179] Khaer Md Abul, Hashem MMA, Raihan Masud Md. An empirical analysis of software systems for measurement of design quality level based on design patterns. IEEE; 2007.
[180] Li Wei, Shatnawi Raed. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. J Syst Softw 2007;80:1120–8.
[181] Kruchten Philippe. What do software architects really do? J Syst Softw 2008;81(12):2413–6.
[182] Khomh Foutse, Gueheneuc Yann-Gael. Do design patterns impact software quality positively? In: Proc.12th conf. on soft. maintenance and reengineering. IEEE; 2008. p. 274–8.
[183] Olbrich S, Cruzes DS. The evolution and impact of code sells: a case study of two open source systems. In: 3rd inter. symposium on empirical soft. engg. and measurement; 2009. p. 390–400.
[184] Singh Satwinder, Kahlon KS. Effectiveness of refactoring metrics model to identify smelly and error prone classes in open source software. ACM SIGSOFT Software Engineering Notes Page 1 March 2012 Volume 37 Number 2. p. 1–11.
[185] Carneiro Glaucode F, Silva Marcos, Mara Leandra, Figueiredo Eduardo, Sant'Anna Claudio, Garcia Alessandro, et al. Identifying code smells with multiple concern views. In: Proceedings – 24th Brazilian symposium on software engineering, SBES 2010 Brazilian symposium on software engineering; 2010. p. 128–137.
[186] Bertran Isela Macia, Garcia Alessandro, von Staa Arndt. An exploratory study of code smells in evolving aspect-oriented systems. In: Conference on Aspect-oriented March 21–25, 2011, Pernambuco, Brazil, ACM.
[187] Luo Yixin, Hoss Allyson, Carver DorisL. An ontological identification of relationships between anti-patterns and code smells. In: Aerospace Conference. IEEE; 2010. p. 1–10.
[188] Fontana Francesca Arcelli, Mariani Elia, Morniroli Andrea, Sormani Raul, Tonello Alberto. An experience report on using code smells detection tools. In: Proceedings – 4th IEEE international conference on software testing, verification, and validation workshops, ICSTW; 2011. p. 450–7.
[189] Macia Isela, Garcia Alessandro, von Staa Arndt, Garcia Joshua, Medvidovic Nenad. On the impact of aspect-oriented code smells on architecture modularity: an exploratory study. In: Fifth Brazilian symposium on software components, architectures and reuse; 2011. p. 41–50.
[190] Singh Satwinder, Kahlon KS. Effectiveness of encapsulation and object-oriented metrics to refactor code and identify error prone classes using bad smells. ACM SIGSOFT Softw Eng Notes 2011;36(5):1–10.
[191] Weber Barbara, Reichert Manfred, Mendling Jan, Reijers Hajo A. Refactoring large process model repositories. Comput Ind 2011;62(5):467–86.
[192] Arcelli Francesca, Spinelli Stefano. Impact of refactoring on quality code evaluation. Copyright 2011 ACM; 2011. p. 37–40.
[193] Izurieta Clemente, Bieman James M. A multiple case study of design pattern decay, grime, and not in evolving software systems. Software Qual J 2012:289–323.
[194] Fontana Francesca Arcelli, Ferme Vincenzo, Spinelli Stefano. Investigating the impact of code smells debt on quality code evaluation. In: 3rd International Workshop on Managing Technical Debt, MTD; 2012 pp. 15–22 – Proceedings.
[195] Ouni Ali, Kessentini Marouane, Sahraoui Houari, Boukadoum Mounir. Maintainability defects detection and correction: a multi-objective approach. Autom Softw Eng 2013;20:47–79.
[196] Derstena Sara, Axelssona Jakob, Froberga Joakim. An empirical study of refactoring decisions in embedded software and systems. Procedia Comput Sci 2012;8:279–84.
[197] Kim Miryung, Zimmermann Thomas, Nagappan Nachiappan. A field study of refactoring challenges and benefits SIGSOFT'12/FSE-20, November 11–16, 2012, Cary, North Carolina, USA. Copyright 2012 ACM 978-1-4503-1614-9/12/11.
[198] Yamashita Aiko, Counsell Steve. Code smells as system-level indicators of maintainability: an empirical study. J Syst Softw 2013;86:2639–53.
[199] Pinto Gustavo, Kamei Fernando. What programmers say about refactoring tools? An empirical investigation of stack overflow, WRT '13, October 27, 2013 ACM.
[200] Balaban Mira, Maraee Azzam, Sturm Arnon, Jelnov Pavel. A pattern-based approach for improving model quality. Softw Syst Model 2013:1–29.
[201] Misbhauddin Mohammed, Alshayeb Mohammad. UML model refactoring: a systematic literature review. Empirical Softw Eng 2015;20:206–51.
[202] Singh Satwinder, Mittal Puneet. Empirical model for predicting high, medium and low severity faults using object oriented metrics in Mozilla Firefox. Int J Comput Appl Technol 2013;47(2/3):110–24.
[203] Dijkman Remco, Gfeller Beat, Kuster Jochen, Volzerb Hagen. Identifying refactoring opportunities in process model repositories. Inf Softw Technol 2011;53(9):937–48.
[204] Wert Alexander, Oehler Marius, Heger Christoph, Farahbod Roozbeh. Automatic detection of performance anti-patterns. In: Inter-Component Communications. ACM; 2014.
[205] Sahin Dilan, Kessentini Marouane, Bechikh Slim, Deb Kalyanmoy. Code-smell detection as a bilevel problem. ACM Trans Softw Eng Methodol 2014;24(1):1–44. Article 6, Pub. date: September 2014.
[206] Mansoor Usman, Kessentini Marouane, Bechikh Slim, Deb Kalyanmoy. Code-smells detection using good and bad software design examples; 2013. p. 1–15.
[207] Mittal Puneet, Singh Satwinder, Kahlon KS. Identification of error prone classes for fault prediction. Berlin Heidelberg: Springer-Verlag; 2011. p. 58–68.
[208] Abebe Mesfin, Yoo Cheol-Jung. Classification and summarization of software refactoring researches: a literature review approach. Adv Sci Technol Lett 2014;46:279–84.
[209] Morales Rodrigo, McIntosh Shane, Khomh Foutse. Do code review practices impact design quality? A case study of the Qt, VTK, and ITK Projects IEEE; 2015. p. 171–80.
[210] Jaafar Fehmi, Gueneuc Yann-Gael, Hamel Sylvie, Khomh Foutse, Zulkernine Mohammad. Evaluating the impact of design pattern and anti-pattern dependencies on changes and faults. Empirical Softw Eng 2015.
[211] Fenske Wolfram, Schulze Sandro. Code smells revisited: a variability perspective. In: Proceedings of the ninth international workshop on variability modelling of software-intensive systems; 2015. p. 3:3–3:10, Germany Copyright 2015 ACM.
[212] Abílio Ramon, Padilha Juliana, Figueiredo Eduardo, Costa Heitor. Detecting code smells in software product lines – an exploratory study. In: 12th international conference on information technology – new generations; 2015. p. 433–8.

[213] 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. In: 2015 12th working conference on mining software repositories. IEEE; 2015.
[214] Bavota Gabriele, Qusef Abdallah, Oliveto Rocco, De Lucia Andrea, Binkley Dave. Are test smells really harmful: an empirical study; 2014. p. 1052–94.
[215] Ounia Ali, Kessentini Marouane, Sahraoui Houari, Inoue Katsuro, SalahHamdi Mohamed. Improving multi-objective code-smells correction using development history. J Syst Softw 2015;105:18–39. Contents.
[216] Kaur Sharanpreet, Singh Satwinder. Influence of anti-patterns on software maintenance: a review, international journal of computer applications. In: International conference on advancements in engineering and technology (ICAET 2015); 2015. p. 14–19.
[217] Chen Jie, Xiao Junchao, Wang Qing, Osterweil Leon J, Li Mingshu. Perspectives on refactoring planning and practice: an empirical study. Empir Softw Eng 2016;21(3):1397–436.
[218] Neto Baldoino 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 Syst Appl 2015;42(3):1652–64.
[219] Khomh Foutse, Di Penta Massimiliano, Gueneucehneuc Yann-Gael. An exploratory study of the impact of code smells on software change-proneness. In: 16th Working Conference on Reverse Engineering; 2009. p. 75–84.
[220] Counsell S, Hierons RM, Najjar R, Loizou G, Hassoun Y. The effectiveness of refactoring, based on a compatibility testing taxonomy and a dependency graph. In: Proceedings of the testing: academic & industrial conference – practice and research techniques. IEEE; 2006.
[221] Bertran Isela Macia, Detecting architecturally-relevant code smells in evolving software systems, ICSE'11, May 21–28, Waikiki, Honolulu, HI, USA ACM; 2011. 978-1-4503-0445-0/11/05.
[222] Anbalagan Prasanth, Xie Tao. Automated inference of pointcuts in aspect-oriented refactoring. In: 29th international conference on software engineering (ICSE'07) 2007.
[223] Marticorena Raul, Lopez Carlos, Crespo Yania, Javier Perez F. Refactoring generics in JAVA: a case study on Extract Method, 2010 14th European conference on software maintenance and reengineering.
[224] Rizvi SAM, Khanam Zeba. A methodology for refactoring legacy code. IEEE; 2011.
[225] Lopez-Herrejon Roberto E, Montalvillo-Mendizabal Leticia, Egyed Alexander. From requirements to features: an exploratory study of feature-oriented refactoring. In: 15th international software product line conference IEEE. IEEE; 2011. p. 181–90.
[226] Lee Da Young. A case study on refactoring in haskell programs. ICSE'11, ACM; 2011. p. 1164–6.
[227] Halim Arwin, Mursanto Petrus. Refactoring rules effect of class cohesion on high-level design. 978-1-4799-0425-9/13/$31.00; 2013 [IEEE].
[228] Nguyen Hoan Anh, Nguyen Hung Viet, Nguyen Tung Thanh, Nguyen Tien N. Output-oriented refactoring in PHP-based dynamic web applications. In: 2013 IEEE international conference on software maintenance.
[229] Kannangara SH, Wijayanayake WMJI. Impact of refactoring on external code quality improvement: an empirical evaluation. In: International conference on advances in ICT for emerging regions (ICTer); 2013. p. 60–7.
[230] Szoke Gabor, Antal Gabor, Nagy Csaba, Ferenc Rudolf, Gyimothy Tibor. Bulk fixing coding issues and its effects on software quality: is it worth refactoring? In: 14th IEEE international working conference on source code analysis and manipulation; 2014.
[231] Zhao Song, Bian Yixin, Zhang Sen Sen. A review on refactoring sequential program to parallel code in Multicore Era. In: International Conference on Intelligent Computing and Internet of Things (IC1T); 2015 IEEE.
[232] Bois Bart Du, Gorp Pieter Van, Amsel Alon, Eetvelde Niels Van, Stenten Hans, Demeyer Serge et al. A discussion of refactoring in research and practice, Belgium; 2004.
[233] Shrivastava Suprika Vasudeva, Shrivastava Vishal. Impact of metrics based refactoring on the software quality: a case study. TENCON IEEE Conference; 2008. p 1–6.
[234] Basili Victor R, Briand Lionel C, Melo Walcelio L. A validation of object-oriented design metrics as quality indicators. IEEE Trans Soft Eng 1996;22 (10).
[235] Huston Brian. The effects of design pattern application on metric scores. J Syst Softw Issue 2001;58:261–9.
[236] Mantyla Mika. Experiences on applying refactoring. Helsinki University of Technology 2002 HUT SoberIT, T-76.650 Software Engineering Seminar.
[237] Gustafsson Juha, Paakki Jukka, Nenonen Lilli, Inkeri Verkamo A. Architecture-centric software evolution by software metrics and design patterns. In: Proceedings of the sixth European conference on software maintenance and reengineering (CSMR'02). IEEE; 2002.
[238] Baldassari Boris, Robach Chantal, du Bouquet Lydie. Early metrics for object oriented designs. IEEE; 2004.
[239] Srivisut Komsan, Muenchaisri Pornsiri. Bad-smell metrics for aspect-oriented software 6th IEEE/ACIS international conference on computer and information science (ICIS 2007); 2007.
[240] Catal Cagatay, Diri Banu. Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem. Inf Sci 2009;179:1040–58. Elsevier.