Arab J Sci Eng (2017) 42:483–506
DOI 10.1007/s13369-016-2238-8

RESEARCH ARTICLE - COMPUTER ENGINEERING AND COMPUTER SCIENCE

A Lightweight Approach for Detection of Code Smells

Ghulam Rasool[1] · **Zeeshan Arshad**[2]

Received: 14 November 2015 / Accepted: 8 June 2016 / Published online: 6 July 2016

Abstract The accurate removal of code smells from source code supports activities such as refactoring, maintenance, examining code quality etc. A large number of techniques and tools are presented for the specification and detection of code smells from source code in the last decade, but they still lack accuracy and flexibility due to different interpretations of code smell definitions. Most techniques target just detection of few code smells and render different results on the same examined systems due to different informal definitions and threshold values of metrics used for detecting code smells. We present a flexible and lightweight approach based on multiple searching techniques for the detection and visualization of all 22 code smells from source code of multiple languages. Our approach is lightweight and flexible due to application of SQL queries on intermediate repository and use of regular expressions on selected source code constructs. The concept of approach is validated by performing experiments on eight publicly available open source software projects developed using Java and C# programming languages, and results are compared with existing approaches. The accuracy of presented approach varies from 86–97 % on the eight selected software projects.

Keywords Code smells · Code flaws · SQL · Regular expressions · Refactoring

✉ Ghulam Rasool
grasool@ciitlahore.edu.pk

[1] Department of Computer Science, COMSATS Institute of Information Technology, Lahore, Pakistan

[2] Department of Computer Science, Punjab University Gujranwala Campus, Gujranwala, Pakistan

1 Introduction

According to Martin Fowler, "a code smell is a surface indication that usually corresponds to a deeper problem in the system" [8]. Code smells are symptoms of poor coding practices adopted by developers during development of software applications due to deadline pressure, lack of awareness and inexperience of developers. Fowler et al. [8] identified 22 code smells with the aim to indicate software refactoring opportunities. The key motivation for detecting code smells is refactoring of source code and to reduce the maintenance efforts. Refactoring is recommended if internal structure of software has symptoms which make it hard to comprehend and evolve. It is reported in different studies that software maintenance cost is around 70–80 % of total software cost [5]. The impact of code smells on maintenance efforts remained debatable topic in the last decade [1,10,20,53,54]. Empirical studies on code smells also showed that code smells hinder comprehensibility and they negatively affect maintainability of software systems [1,54]. A large number of techniques and tools are presented for the specification, detection and visualization of code smells in last one and half decade.

We can classify state-of-the-art code smell detection techniques into two general categories. The first type concerns qualitative detection using (inevitably biased) expert-based heuristics. The second type applies threshold values on software metrics. This category also uses combination of code metrics, source code parsing, and historical information and seems more appealing. The techniques based on software metrics have limitations that same set of metrics cannot be used to detect all smells of Fowler's catalog [8] due to the reason that code smells are very distinct in nature. Mantyla el al. [12] discussed the difficulty of assessing code smells by using code metrics. A concept of relative threshold values

is proposed by Oliveira et al. [15] to create balance between real and idealized design rules that are used to access quality of object-oriented systems. The source code parsing-based techniques scan source code and use static/dynamic analysis to extract information directly from the source code for detection of code smells. Some techniques apply combination of metrics and source code parsing for detection of smells with improved accuracy. The unique concept of historical information for detection of code smells from source code is adopted by different code smell detection techniques [9,16,17,19]. The key motivation was to detect smells which cannot be detected by using other techniques.

Through review of the literature, we found issues with state-of-the-art code smells detection techniques that: All presented techniques focused on subset of 22 code smells during recognition of smells from the source code. To the best of our knowledge, there is no technique that detected all 22 smells from the source code of multiple languages. Secondly, there is still no consensus on standard definitions and detection algorithms that are used by different techniques. Due to this reason, there is a disparity in the results of different techniques on the same examined systems. Thirdly, most techniques focus on a single language for code smells detection. Most techniques focused on Java language for detecting code smells. The multiple-language support for the detection of code smells is still limited. Fourth, existing taxonomies on code smells categorization are not capable enough to group all code smells for their better comprehension. Finally, there is still no standard benchmark corpus for evaluating results of code smells detection techniques and tools.

The motivation of our approach is to address above-mentioned issues related to existing code smells detection techniques. We provide flexible definitions and detection algorithms for the recovery of all 22 code smells from source code of Java/C# languages. Our approach is first time used to detect all 22 code smells from source code of C# language. The proposed approach is customizable and extendable for other languages due to support of Add-In [21] for reverse engineering of 14 programming languages. Our approach is lightweight as it uses SQL queries and regular expressions as searching mechanisms for the detection of code smells as compared to techniques that apply hard coded algorithms [3,7,52]. SQL queries are independent of programming language constructs, and same queries may be reused for detection of similar smells and smells from other languages. The regular expression patterns are also flexible and easy to write for recovering information from source code directly that was not available in the intermediate representation of the source code. However, regular expressions are dependent on a specific programming language. Due to limitations of existing taxonomies [11,13,14] on code smells discussed in Sect. 3, we present a new taxonomy for better comprehension of code smells. Our taxonomy is capable to group all existing as well as new code smells. We provide initial catalogue for the results of all 22 code smells first time that can be used by researchers for comparing results of code smell detection techniques. The unique contributions of our approach are as follows:

- A new taxonomy for categorization of Fowler's 22 code smells;
- Flexible definitions and specifications for all 22 code smells;
- Multiple-language support for the detection of all 22 code smells;
- A prototyping tool for implementation of proposed approach;
- A benchmark system for evaluation and comparison of results for different tools.

The rest of paper is structured as: Sect. 2 discusses state of the art in the field of code smells detection and highlights limitations of existing techniques. A new taxonomy for code smells is presented in Sect. 3. The specifications for three selected code smells are described in Sect. 4. The code smells detection approach is presented in Sect. 5. We evaluate our approach and compare its results with state-of-the-art approaches in Sect. 6. Finally, Sect. 7 discusses the conclusion and future work.

2 State of the Art

Research on code smells detection has been started since 1999 when Fowler et al. [8] identified code smells with the aim of software refactoring. A number of techniques have been presented for definitions, specifications, detection methods and better understanding of code smells. Mentioning all techniques here is superfluous since many reviews and comparisons have been conducted in the past. We presented a recent review on code smells mining techniques that covers different aspects of code smells [18]. A review in 2011 [55] analyzed 39 studies relevant to the code smells understanding, detection and their impacts on software maintenance. Authors concluded that impact of all code smells has not yet been fully understood. Main reason is the varying definitions of understanding for code smells and the varying nature of smell detection tools. Most tools are capable of recognizing code smells from the source code of one language, and they focus on few smells. We discuss selected prominent approaches in this section and present a summary about discussed approaches in Table 1.

The detection of code clones (Duplicate Code) is the code smell that got maximum attention of researchers [55]. We exclude approaches that focus only on single Duplicate Code smell. Marcus et al. [22] presented an approach that examines

Table 1 Summarized information about code smells detection techniques

References	Tool name	Detection approach	Supported languages	Detected smells	Study type and evaluation method	Accuracy
Marcus and Maletic [22]	N/A	Latent semantic indexing	C	Cmnt, DupC	Experiments on Mosaic (v2.7)	NM
Marticorena and Crespo [24]	An eclipse plug-in	Software metrics	Java	PIH	Case study on JFreeChart, jcoverage and JUnit	NM
Munro [26]	N/A	Combination of metric and interpretation rules	Java	LzyC, TF	Case study on a commercial project	NM
Mihancea and Marinescu [25]	N/A	Tuning machine (method)	Java	DC, GC	Experiment based on student projects	NM
Slinger [35]	CodeNose An eclipse plug-in	JDT parser	Java	LM, LPL, FE, MC, SSC, RB, LzyC	Experiments on JHotDraw5.3	NM
Tsantalis [52]	jDeodorant	ASTParser	Java	FE, TC	Experiment on open source(video store, LAN simulation etc)	$P = 92\,\%$
Chaikalis et al. [3]						
Marinescu and Ratiu [23]	Prodeos	Metric based	C++, Java	GC, SS, RB, DC	Case study on two industrial telecommunication software systems	NM
Naveen [27]	JSmell	Source code parsing	Java	DC, MC, PO, SG, PIH, DupC Cmnt	Test case: tested against seven different version of a system	$P = 85 - 90\,\%$
Moha et al. [14]	Detex	Based on DECOR	Java	MC, SS, DupC, DC, LC Cmnt, NP, GV,CC,PC, LM,, NI, LoCo, DivC, LPL	Experiment on 11 different systems: PMD, ARGOUML, ECLIPSE QUICKUML etc	$R = 100\,\%$ $P = 50\,\%$
Slinger [28]	Java Smell Detector (JSD)	Statistical data gathering technique	Java	SSC, DC, MM, LPL and LM	test case: tested against 28 graduate projects	NM
Nongpong [33]	JCodeCanine (Eclipse plug-in)	Integration of metric-based approach and detailed analysis	Java	DupC, FE, DC, SSC	Experiments on 7 real world projects. 3 Fluid Framework and 4 students projects	54 % average accuracy
Gopalan [34]	PMD Extended Rules	Software Metrics with new rule sets	Java	LC, LM, LPL, PO,DaC	Experiments on ArgParser, jCrawler, jLayer etc	NM
Fard and Mesbah [6]	JSNOSE	Combination of Metrics with static and dynamic analysis	JavaScript	JavaScript smells (thirteen)	Investigated 11 web applications	$P = 93\,\%$ $R = 98\,\%$

Table 1 continued

References	Tool name	Detection approach	Supported languages	Detected smells	Study type and evaluation method	Accuracy
Palomba et al. [16]	Research prototype	Change history information extracted from versioning systems	Java	DivC, SS, PIH, Blob, FE	Case study on eight java open sources, e.g., Apache Ant, Apache Tomcat, jEdit etc	NM
Sahin et al. [36]	Prototype tool (BLOP)	Bi-level Optimization	Java	FE, DC,LC, LPL and 3 design smells	Experiments on 9 open source systems	$P = 85\,\%$ $R = 90\,\%$
Kim e al. [37]	An eclipse plug-in	OCL component (JavaEAST model)	Java	LPL, SSC, TF, MC, FE, RB	Experiments on small examples	NM
Kessentini et al. [38]	Research prototype (P-EA)	Co-operative based	Java	DC, LYz, LPL, FE, SS and 3 design smells	Experiments on large open source systems	$P = 85\,\%$ $R = 85\,\%$
dos Santos Neto [39]	JDeodorant, CheckStyle and inCode	Multiple techniques	Java	DaC, FE, PF	Experiments on five open source systems	NM
Palomba et al. [17]	Research prototype	HIST and competitive code analysis techniques	Java	DivC, SS, PIH, Blob, FE	Experiments on 20 open source projects	$P = 72 - 86\,\%$ $R = 58 - 100\,\%$
Ganea et al. [40]	inCode	Metrics based	Java	DC, FE, DupC, GC	Experiments on five open source systems	NM

Cmnt Comment, *CC* Controller Class, *DC* Data Class, *DaC* Data Clump, *DupC* Duplicate Code, *DivC* Divergent Change, *FE* Feature Envy, *GC* God Class, *GV* Global Variable, *LC* Large Class, *LoCo* Low Cohesion, *LzyC* Lazy Class, *LM* Long Method, *LPL* Long Parameter List, *MM* Middle Man, *MC* Message Chain, *NP* No Polymorphism, *NI* No inheritance, *PO* Primitive Obsession, *PC* Procedural Class, *PIH* Parallel Inheritance Hierarchy, *RB* Request Bequest, *SSC* Switch Statement Case, *SS* <>Shotgun Surgery, *TF* Temporary Field, *IC* Intensive Coupling, *PF* Public Field, *NM* not mentioned, *P* precision, *R* recall

the source code text (identifiers and comments) to identify these similar high-level concepts (clones) from the source code. They used information retrieval technique to analyze the software system and determine semantic similarities between source code documents. They evaluated their approach also known as "latent semantic indexing" to detect Duplicate Code and Comment code smells from NCSA Mosaic [22].

Nitin and Mathur [28] proposed a Java Smell Detector (JSD) tool for recovering smells from object-oriented Java software projects. JSD is based on the statistical data gathered (Lines of Code, Method Invocation, Method Names etc.) while parsing the Java files. JSD is capable to detect "Switch Cases," "Data Class," "Middle Man," "Long Parameter List," and "Long Method" smells, based on the statistical data detected while parsing through files. It also provides refactoring support for two smells, i.e., "Data Class" and "Long Method." JSD was tested against 28 projects taken from the graduate students of San Jose State University. These selected students have experience of two to five years as Java developers, and the complexity of their code is considerable. Each project has an average of 13 classes. However, we realize that JSD's functionality is limited to Java language and to the detection of five code smells only. Moreover, if JSD provides an interactive refactoring approach for other code smells, it will become a powerful tool.

Mihancea et al. [25] presented a novel method based on tuning machine method to detect design flaws, i.e., "God Class" and "Data Class" from object-oriented projects. The genetic algorithms are used for establishing proper threshold values for metric-based rules used to detect these smells. Authors performed experiments on small student projects, and they did not evaluate the accuracy of applied technique.

Moha et al. [14] proposed code smells detection technique DETEX that is an instantiation or a concrete implementation of DÉCOR method. DÉCOR is a concrete and generic method for the detection of code smells. Authors applied DETEX on four design smells and 15 code smells [14]. Validation is measured by experimenting tool on 11 object-oriented software projects. Authors use automatically generated detection algorithms, and they have recall of 100 %, i.e., all known design smells are detected. However, the precision of applied approach is approximately 50 %, i.e., the detection algorithms are better than random chance. Authors mentioned that improvement in detection method and algorithms can be made by comparing with existing tools and by applying detection technique to other types of smells as well [14].

Munro [26] proposed an approach based on the combination of software metrics for detecting code smells directly from the source code. To justify the choice of metrics and thresholds for detecting smells, he performed an empirical study. However, his work is limited to the identification of only two code smells, i.e., Lazy Class and Temporary Field. Metrics is an Eclipse plug-in to detect "Parallel Inheritance Hierarchy" code smell developed by Marticorena et al. [24]. Authors used different metrics such as Depth Inheritance Hierarchy (DIT) and Number of Children (NOC) in their plug-in to detect Parallel Inheritance Hierarchy code smell from source code of Java language only [24].

Prodeos is a smell detection tool [23] that detects design flaws from the source code of Java and C++ programming languages. The strategy applied by Prodeos is based on software metrics that are analyzed through the statistical data captured while parsing through the source code. In the end result, a report is generated that contains all the design flaws. Although Prodeos is capable of detecting smells from C++ and Java source code, its detection capability is limited to few code smells as indicated in Table 1. "Feature Envy" and "Type Checking" are the code smells basically detected by JDeodorant tool [3,52]. JDeodorant uses the ASTParser API of Eclipse to detect the code smells from the source code. The tool is mature enough and freely available to perform experiments. Authors used the concept of measuring distance between classes, methods and attributes for the detection of Feature Envy code smell and Move method refactoring opportunity for its eradication [52].

JSmell [27] is a smell detector tool developed in C# language for recovering code smells from Java source code. It detects seven code smells: "Data Class," "Message Chain," "Primitive Obsession," "Speculative Generality," "Parallel Inheritance Hierarchy," "Duplicate Code" and "Comments." Authors used ANother Tool for Language Recognition (ANTLR) parser to parse the code files and gather the statistical results to classify the smells. JSmell has two phases to identify smells. During the first phase, it parses all the Java source code files and gathers required data such as method declaration, variable declaration and class names. In the second phase, it uses this statistical data and parses all the code again to identify the smells present in each of them [27]. The success rate for code smell detection process is 85–90 % when tested against seven different test cases for a single version of a system. JSmell detector also performs structural analysis on the source code in terms of classes, methods and data fields that facilitates the developers in understanding the high-level system architecture.

Fard et al. [6] presented a JavaScript code smell detection technique called JSNOSE. They detected 13 JavaScript code smells. Out of 13 code smells, 7 are existing well-known smells adapted to JavaScript, and 6 are specific JavaScript code smell types, collected from various JavaScript development resources. JSNOSE uses a metric-based approach that combines static and dynamic analysis to automatically detect smells in the client-side code. This automated technique can help developers to spot code that could benefit from the refactoring. By analyzing 11 web applications that

make extensive use of client-side JavaScript and fall under different application domains, they investigate which smells detected by JSNOSE are more prevalent. Results of 9 applications (other 2 are by code large in size) show that JSNOSE has an overall precision of 93 % and an average recall of 98 % which reflect its effectiveness. The current implementation of JSNOSE is not able to detect various ways of object creation in JavaScript. JNOSE also does not deal with various syntax styles of frameworks such as jQuery [6].

Nongpong [33] presented an approach that combines code smells detection and refactoring. A tool for Java programs called JCodeCanine has been developed in which code smells detection and refactoring are connected. JCodeCanine is implemented based on existing components which are Fluid and Eclipse. The Fluid's infrastructure is used as the main component for back-end process and Eclipse is used mainly for the user interface. JCodeCanine detects code smells within a program based on metrics and source code analysis. Author proposed a list of refactoring options that help to improve the internal quality of software. The programmer has an option whether to apply the suggested refactoring through "quick fix." Author himself pointed out that source code analysis method can be used in addition to metrics which results in more precise results. He evaluated JCodeCanine on 7 real-time software projects.

A unique approach [16] named as HIST (Historical Information for Smell detection) is presented to detect five different code smells. Authors detect code smells based on change history information extracted from versioning systems and specifically by analyzing co-changes occurring between source code artifacts. Authors applied HIST approach on the eight software projects written in Java (namely Apache Ant, Apache Tomcat, JEdit, and five projects belonging to the Android APIs) and wherever possible compared with existing state-of-the-art smell detectors based on source code analysis. The results indicate that HIST's precision ranges between 61 and 80 % and its recall ranges between 61–100 %. More importantly, the results confirm that HIST is able to identify code smells that cannot be identified through approaches solely based on source code analysis [16]. The main limitation of HIST is represented by the need for having sufficient history of observable co-changes, without which the approach falls short. Authors presented extended approach based on same concepts and improved accuracy for detecting five code smells. The new approach is evaluated using two empirical studies. In a first study, they performed experiments on 12 open source projects and compared their results with state-of-the-art code smell detection tools. In a second study, authors involved 12 developers of four open source projects and concluded that 75 % of smells detected in first study are also recognized as design and implementation problems by developers.

Gopalan [34] extended the existing open source code smells detection tool PMD by adding new rule sets using XML specification language XPATH. Author tested new rule sets on different open source software projects. The extended approach is able to detect more code smells that was not possible with the default rule set. Slinger [35] in his master thesis presented a prototyping tool JCodeNose to detect seven different code smells from JHotDraw 5.3. Author constructed AST of sourced code using Eclipse JDT parser. A tree visitor collects different aspects related with smells that are used for the detection of code smells.

Sahin et al. [36] presented an approach that takes code smell detection problem as bi-level optimization problem. The upper level generates detection rules, and lower level maximizes chances of code smells that are not detected in the first level. The key advantage of approach is not only to detect code smells already defined, but also to predict behavior that is different from the base example. Authors performed experiments on the eight open source software projects and concluded that bi-level optimization outperforms state-of-the-art code smell detection techniques.

Kim et al. [37] specified and detected code smells by using OCL (Object Constraint Language). By running OCL components, they precisely detect code smells automatically. The OCL code model runs through OCL component based on Eclipse plug-in for auto detection of code smells. Authors transform Java source code into a format known as XMI (XML Metadata Interchange). This format is based on expanded JavaAST meta-model known as JavaEAST (Java Extended Abstract Syntax Tree) meta-model.

Kessentini et al. [38] presented a search-based approach using Parallel Evolutionary Algorithms (P-EA) for the detection of code smells. They integrated different methods in parallel and detected 8 code smells from eight large open source software projects. Authors applied two parallel algorithms which speed up search process and reduce search space. The first algorithm generates detection rules, and the second algorithm generates detectors. Both algorithms are based on genetic programming. Generalization of approach for detection of other types of code smells is questionable.

Authors presented an agent-based framework that is used for automatic refactoring of the source code [39]. The unique features of approach are identification of code smells, determination of corrections, assessment of quality and preservation of observable behavior. The application of applied technique ensures that quality of source code is improved after performing refactoring operation. The experiments are performed on different open source projects to evaluate the accuracy of approach. Ganea et al. [40] presented a code smell detection technique supplemented with a tool support inCode for the source code analysis and quality assessment. The major focus of authors is to detect and correct different design problems. The tool has support for Eclipse plug-in,

and it warns developers about occurrence of design problems when they appear in the source code. Authors performed experiments on different open source projects and detected four code smells.

We summarize from state of the art that most techniques apply existing/new object-oriented source code metrics for detection of code smells. The same set of metrics cannot be used to detect all code smells [8]. A few techniques used source code parsing techniques for the detection of code smells. A number of techniques also apply the combination of code metrics and source code parsing for detection of code smells with better accuracy. We also realized that most techniques focused on detection of code smells from Java source code and they perform experiments on small subset of code smells. Long Method, Long Parameter List and Data Class smells gained maximum attention of researchers according to Table 1. Furthermore, accuracy of code smells detection techniques is a key aspect for their validity, but many authors did not mention the accuracy of their results.

3 Taxonomy of Code Smells

Categorization of code smells enhances the understanding in larger context as compared to individually understanding of each smell from a flat list of 22 code smells [47]. The first taxonomy on code smells is presented by Mantyla et al. [47] that is based on common concepts that the smells share inside one group. Authors mapped code smells to 7 higher-level categories. This taxonomy was preliminary and had limitations that need to be improved. Three years later in 2006 [11], the authors themselves introduced a refined taxonomy to suppress limitations of their previous classification. They mapped 20 smells of Fowler and one other that is "Dead Code" in five distinct groups. The new classification fails to list all 22 smells (identified by Fowler) in distinct groups. For example, "Incomplete Library Class" and "Comments" are not suitable to fit in any group. In their previous classification [47], they grouped these two smells in the category of "Others." Later on, Marticorena et al. [13] extended his taxonomy by adding different metrics such as granularity of smells (System, Class or Method Level), requirement of inter or intra-relation of component (between classes or methods), inheritance and access modifier requirement for smell detection. However, they [13] also did not succeed to adjust "Incomplete Library Class" and "Comments" in any group. Moha et al. [14] presented a taxonomy for design smells which includes some code smells of Fowler's list. The key focus of authors was to classify design smells for their detection. We present a comparative overview of features of taxonomies discussed above in Table 2.

The motivation for introducing a new taxonomy for code smells is to overcome limitations of existing taxonomies for better understanding and grouping of code smells. Our proposed classification is an attempt to fit all 22 smells (including "Incomplete Library Class" and "Comments") in their specific groups. The proposed taxonomy can accommodate all 22 code smells of Fowler and new code smells. Our taxonomy classifies 22 smells in five groups as given below in Table 3. These groups are based on the common concepts that smells have inside a group, but at more abstract level, to fit all 22 smells along with some other found in the literature. Comments in the source code are used for descriptions of programmer's intentions behind the source code. Comments can be at higher level of abstraction than code, and they are helpful for the comprehension of source code. They also reflect discipline of developers followed in the source code. We placed Comments code smell in structure-based group due to the fact that comments have impact on the structure of the source code. The importance of source code comments regarding structure and comprehension of source code is also realized by Khamis et al. [43], but excessive use of comments makes reading of source code boring. The presence of excessive inline comments show that source code needs refactoring. Incomplete Library Class code smell occurs when a built-in library or third-party components cannot meet requirements of developers. It shows that library is incomplete or some features of library are missing that are required. We placed that smell in the category of feature-based group. Our proposed taxonomy is able to group all 22 code smells, and it is extendable for other code smells. Moreover, we think it is more comprehensible and generic as compared to existing taxonomies.

4 Specification of Code Smells

The accuracy of code smells detection approaches depends on the accurate specification of code smells. Due to variations in the specifications of code smells by different authors, we present our specification of all code smells used by our code smell approach in this section. Another motivation of our own specifications for code smells is to let the user customize specifications of code smells in order to handle different variations. We undertake definitions of code smells presented by Fowler et al. [8] and other authors [7,14] for writing specifications. For example, the specifications of Long Method and Large Class code smells used in our approach are similar to specifications used in PMD and CheckStyle. We allow user to set threshold values while detecting these code smells. Most of these specifications are fine enough to detect the given smell, but they vary in detection steps. We tried to build our specifications based on existing specifications used in different tools instead of reinventing the wheel. However, specification of some code smells in our approach is relatively different or even entirely unique as compared to the

Table 2 Comparative overview of Taxonomies

References	Technique	Number of groups	Classify all Fowler's smell	Classify additional smells other than Fowler's
Mantyla et al. [47]	Based on characteristics of smells	Seven	No (only 20)	No
Mantyla et al. [11]	Based on refined [13] characteristics of smells	Five	No (only 20)	Yes (dead code)
Marticorena et al. [13]	Extended [11] by introducing additional metrics	Variable groups (Based on given metrics)	No	No
Moha et al. [14]	Structural, lexical and measurable characteristics	Two	No, focus on design smells	Yes (only few)
Our taxonomy	Based on common concepts	Five	All 22	Yes

Table 3 Taxonomy of code smells

Group name	Objective	Smells
Mass-based categorization	Group smells on the basis of exceptionally larger or immensely smaller sizes of objects, methods and parameters	(i) Long Parameter List
		(ii) Long Method (Brain method/God method)
		(iii) Large Class (God Class)
Feature-based categorization	Group smells on the basis of incomplete, inappropriate or missing behavior	(i) Data Class
		(ii) Middle Man
		(iii) Speculative Generality
		(iv) Divergent Change
		(v) Shotgun Surgery
		(vi) Incomplete Library Class
		(vii) Lazy Class
Dependency-based categorization	Group smells on the basis of coupling or duplication	(i) Duplicated code
		(ii) Switch Statement
		(iii) Parallel Inheritance Hierarchy
		(iv) Feature Envy
		(v) Message Chain
		(vi) Inappropriate Intimacy
Structure-based categorization	Group smells on the basis of inappropriate composition or arrangement of code parts	(i) Temporary Field
		(ii) Refuse Bequest
		(iii) Data Clump
		(iv) Primitive Obsession
		(v) Comments
Name-based categorization	Group smells on the basis of inappropriate name	(i) Uncommunicative name
		(ii) Alternative Classes with Different Interface
		(iii) Inconsistent Name

existing smell detection techniques. For example, specification steps for Long Parameter List, Inappropriate Intimacy and Comments code smells are more accurate and efficient as compared to existing specifications. In order to curtail space, we list specifications of all code smells in "Appendix" and on the web for readers [49]. Here, we discuss only specification of Inappropriate Intimacy, Comments and Long Parameters List code smells.

4.1 Inappropriate Intimacy

Inappropriate intimacy smell exists at:

1. Class level if a calling class C1 calls more than or equal to a specified number of distinct methods of class C2 or
2. method level if a method M of class C1 calls more than or equal to a specified number of times the distinct methods of class C2.

We specify the above steps as:

$MCC2(C1) \geq nMiC\ OR\ MCC2(C1M) \geq nMiC$
where MCC2(C1) = NumberofMethodsofClassC2 called by Class C1
nMiC = NumberofintimatedMethodsinaClass
MCC2(C1M) = MethodsofClassC2 called by Method M of Class C1.

4.2 Comments

Comments are considered as code smell if a class contains:

1. More than or equal to specified percentage of commented code (as compared) to its total lines of code where commented code includes inline comments, single-line comments and multi-line comments.

We specify the above step as:
$LOCCmntd/LOCTotal \geq specified\%$
where LOCCmntd = CommentedLineofCode in a source code class.

4.3 Long Parameter List

A Long Parameter List smell exists if :

1. A method contains more than or equal to a specified number of parameters OR
2. a method contains two or more (specified numbers of) parameter of same primitive data type.

We specify the above steps as:
$PL \geq PLV\ OR\ RDT >> 1$
where PL = ParameterList, PLV = Value for Maximum Number of Parameters, RDT = ReplicatePrimitiveDataType

The definitions of code smells presented in our approach are flexible which give user a choice to customize definitions. For example, the detection of Long Parameter List code smell varies upon number of parameters used by different approaches for the detection of that smell. We provide flexible code smell definitions that allow the user to select number of parameters and threshold values for the detection of different code smells. The complete definitions of all code smells are presented in "Appendix." We also publish definitions for code smells on the web for readers [49].

5 Code Smells Detection Approach

The presented approach is based on the methodology presented by [50] for design patterns detection, but this proposed approach is used for detection and visualization of code smells. We apply integration of source code metrics and source code parsing for the detection of code smells with improved accuracy. The source code of an examined application is reverse engineered using Sparx System Enterprise Architect Modeling tool (EA) [21] which has support for reverse engineering of 14 programming languages. The motivation for selection of this particular reverse engineering tool is due to our prior experience of using the tool for different other projects [32,50]. We get intermediate representation of the source code at an abstract level. The intermediate representation contains database model, UML models and XML representation of the source code. The intermediate representations contain packages, classes, methods, attributes, and other elements as well as association, dependency and inheritance relations among these elements. The EA tool also creates summary of source code metrics, e.g., number of objects, attributes, operations and other information which is helpful for examining initial statistics of source code elements. The key benefit of reverse engineered model is that it contains the structure of source code in query-able repository which allows the efficient searching within the structure of the source code and to match key structural concepts. We used SQL queries to extract information available in the intermediate representation related to code smells. The limitations of the models are unavailability of information such as runtime, versioning, comments, annotations. These limitations motivated us to perform analysis on the source code directly using regular expressions. The selection of regular expression technique for matching information related to code smells is due to easy customization of regular expression patterns for handling variations while recognizing code smells. Regular expressions are powerful for matching constructs from source code directly, but they have low accuracy while extracting nested information from source code.

The architecture of presented approach is given in Fig. 1. The approach is implemented with the help of a prototyping tool. The prototyping tool in large parts is independent from specific modeling tools, and it can be integrated with tools such as IBM Rational Software Modeler, Borland Together. The key components of the approach are code smell definitions, SQL queries and Regular expressions. We explain these components in the following subsections.

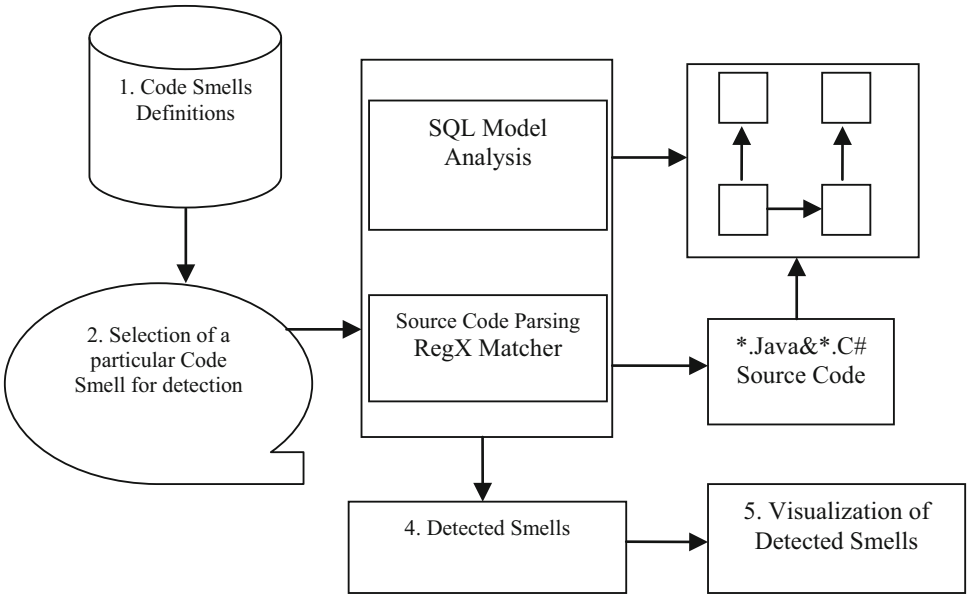

Fig. 1 Architecture of proposed approach

5.1 Code Smell Definitions

The code smell definitions are backbone for accurate detection and removal of code smells from the source code of different applications in our presented approach. We provide an evolvable and customizable code smell definitions catalog containing all commonly agreed code smell definitions available in the literature. The code smell definitions used in our approach are customizable by the user due to option of setting dynamic threshold values at the time of their execution. The definitions are stored in a separate file which makes their updating and extension easy for the user. Each code smell definition is supported with SQL queries or regular expression patterns for detecting code smells from the source code. We present definitions of code smells on the web and "Appendix."

5.2 SQL Queries

The application of SQL queries depends on the prior step of reverse engineering all source code into a source code model. Reverse engineering of source code for large systems is a time-consuming process, but full transformation of source code is necessary only once for a given state of the implementation as modeling tools are able to incrementally update the model with changes to the source code. The queries are independent from a specific programming language, and they are easily customizable for updates. The benefit of queries is that they are relatively simple to write and they are very efficient. SQL databases are established standards that are adopted by ANSI and ISO. It is worthwhile to mention that queries on the repository are more efficient than analyzing the source code directly because the source code model contains typed lists of elements that reduce the search space for structural features dramatically. Furthermore, source code model queries are independent from the programming language of the underlying source code, because general constructs are queried. User can write new queries or update existing queries to detect variations in similar code smells. However, user requires knowledge about internal structure of the database model prior to writing queries. The following example shows a sample SQL query that is used for detection of Long Method code smell.

Example 1: SQL query

"SELECT t_object.Name as Class_Name,t_Operation. Scope+′/′ + t_Operation.Type + ′/′ + t_Operation.Name as Method_Name,t_object.GenFile as Class_Path FROM t_operation INNER JOIN t_object ON t_operation.Object_ID= t_Object.Object_ID WHERE t_Object.Object_Type='Class' AND t_Object.Author IS NOT NULL AND t_Operation. Scope IS NOT NULL AND t_Operation.Type IS NOT NULL AND t_Operation.Name IS NOT NULL GROUP BY t_object.Name,t_object.GenFile,t_operation.Name, t_Operation.Scope,t_Operation.Type Order by t_object. Name".

The above SQL query extracts list of classes, their methods and class path from EA data model. This information is further used to extract Long Method code smell. Extraction of such information directly from source code requires much effort. Similarly, SQL query in Example 2 is used to extract total number of methods in the names of each file. This information is further used to detect Large Class and God Class code smells.

Example 2: SQL query

"SELECT t_object.Name as Class_Name, Count(t_ Operation.Name) as No_of_Methods, t_object.GenFile as Class_Path From t_operation INNER JOIN t_object ON

t_operation.Object_ID = t_Object.Object_ID WHERE t_Object.Object_Type = 'Class' AND t_Object.Author is not null GROUP BY t_object.Name, t_object.GenFile Having COUNT(t_operation.NAME)≥" + paramValue + " Order by t_object.Name".

5.3 Regular Expressions

The key motivation of using regular expressions is to directly extract information from source code and to keep our approach lightweight. We only used regular expression patterns for the extraction artifacts that were not available in the source code model. Mens et al. [48] also used concept of regular expressions for expression of code smells and their detection. They consider regular expressions as more efficient and lightweight approach to compute smells.

The regular expression patterns in our approach are separate from code smells detection algorithms. They are stored in a separate file, and they can be updated easily without changes in the detection algorithms. The regular expressions can use results of queries as arguments if required, and queries can use results of regular expressions. The approach is using regular expressions to extract information either from the source code model or directly from the source code. We apply regular expressions on single-line code constructs, classifier names, comments and annotations. Regular expressions are simple to write and understand while avoiding the comparable high effort for implementing a parser module. Furthermore, regular expressions match in most cases faster than a source code parser. However, regular expressions cannot be used for all matching tasks. A limitation of regular expressions is that they are not able to certainly extract nested information within source code elements. All regular expression editors do not support non-greedy matching. The limitations of regular expressions for source code parsing are also highlighted by authors [31,41]. Example 1 below represents a sample regular expression pattern.

Example 1: RegX

@"(?<body>\{(?<DEPTH>)(?>(?<DEPTH>)\{ |\}(?<-DEPTH>) |(?(DEPTH)[^\{\}] |))*\}(?<-DEPTH>)(?(DEPTH)(?!)))";*

The above pattern indicates how we use regular expression patterns to select the body of methods with their full depth from currently selected portion of the class. Once the body of a method is selected, we can count the number of lines in the method. Similarly, Example 2 presents regular expression pattern that is used to match single-line and multi-line comments from the source code.

Example 2: RegX

(/([^*]|[\r\n]|(*+([^*/]|[\r\n])))**+/)|(//.*)*

5.4 Code Smells Detection Example

We explain our code smells detection approach by an illustrative example. The source code of an examined application is reverse engineered, and we get the intermediate representation in the form of a database model and a class model. Figure 2 presents the sample resulting class model containing six elements and different relationships among them. This example demonstrates how we detect Comment class code smell by using our approach. The detection process follows the following steps:

1. Extract all Classes and Interfaces

This step returns all the Classes and Interfaces from a database model by using a SQL query and we get A, B, C, D, F and G.

2. Calculate percentage of commented number of lines of code and total line of codes for each class.

The regular expression pattern is used to calculate the ratio of comments for all classes mentioned in the above step.

3. Compare percentage obtained in above step with the specified threshold.

The last step returns only classes B, C and G which fulfill our definition of Commented class code smell. The remaining classes are filtered out. The result of third step is extracted from results of step 1 and 2 sequentially.

The source code snip in Fig. 3 is used to implement the above-mentioned three steps used for the detection of commented classes by our prototyping tool. The steps in Fig. 3 are explained below:

Line 1: Indicates SQL query that is used to get data, i.e, the list of classes and their class path from the EA data model.

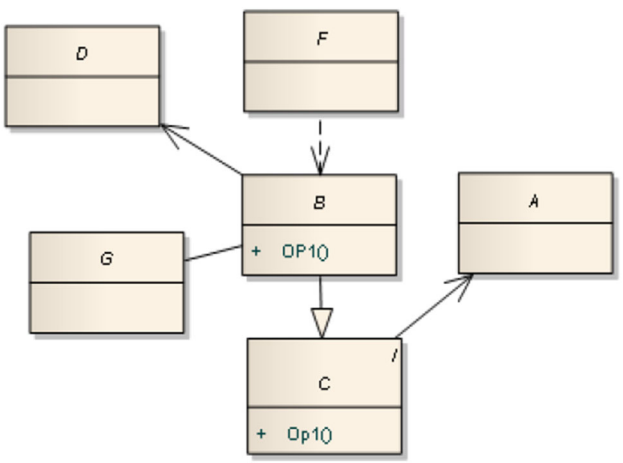

Fig. 2 Illustrative detection example

```
1) {String xmlResult = ex.SQLQuery("SELECT t_object.Name
   as Class_Name,t_object.GenFile as Class_Path FROM t_operation
   INNER JOIN t_object ON t_operation.Object_ID=
   t_Object.Object_ID
   WHERE t_Object.Object_Type='Class' AND t_Object.Author is not
   null
   Group by  t_object.Name,t_object.GenFile");
2) for (int i = 0; i < ds.Tables["Row"].Rows.Count; i++)
3) { String className =
   ds.Tables["Row"].Rows[i]["Class_Name"].ToString();
4) String filePath =
   ds.Tables["Row"].Rows[i]["Class_Path"].ToString();
5) double CmntdLines =
   helpingClass.getClassCommentedLines(className, filePath);
6) double totalLines = helpingClass.getClassTotalLines(className,
   filePath);
7) if (CmntdLines >= totalLines * (double.Parse(thresholdValue) /
   100))
8) dt.Rows.Add(ds.Tables["Row"].Rows[i]["Class_Name"].ToString(),
   totalLines, CmntdLines, Math.Round(((CmntdLines / totalLines) *
   100), 2) + "%",
   ds.Tables["Row"].Rows[i]["Class_Path"].ToString()); }}
```

Fig. 3 Source code snip for detection of Comment code smell

Lines 2–6: Indicates how body of each class is scanned from the path provided by SQL query and how the independent methods for the counting of commented lines and total lines in each class are extracted.

Lines 7–8: Finally, once the commented lines and total lines in a class are obtained, we compared the resultant percentage (class commented lines to class total lines) with the given threshold percentage to decide about existence of Comment code smell.

5.5 Prototyping Tool

We developed a prototyping tool named as multiple language smells Detector (MLSD) for validating our proposed approach. The prototype is implemented using C# and the Microsoft.Net framework. The tool is independent from any reverse engineering and modeling tool. It can be easily integrated with a number of tools such as Borland Together, IBM Rhapsody and IBM Rational Rose Modeler. We selected Sparx System Enterprise Architect Modeling tool for implementation of our prototype due to its excellent capability for reverse engineering source code of 14 programming languages and due to our prior experience in creating extensions for that tool [32,50]. Enterprise Architecture is an agile, cost-effective modeling, design and management platform that accelerates and integrates software, business and systems development. Currently, our prototyping tool is capable of extracting all 22 code smells from the source code of Java and C#, but it is extendable for other languages due to support of EA for reverse engineering source code of 14 programming languages. To the best of our knowledge, there is no freely available code smells detection tool that is capable of detecting 22 code smells from source code of these two languages. InFusion [42] has support for the detection of 18 code smells, but it is commercial tool and is not freely available for evaluation. A comparison of code smells detection tools is difficult due to the reason that different tools are developed for different environments, smells and languages. We compared features of our prototyping tool with CheckStyle [4] and PMD [29] in Table 4. The major reason for the selection of these tools is their free availability and support for the detection of common subset of code smells. We compared results of our tool with these tools in the evaluation section.

Table 4 Comparison with PMD and checkStyle

Features	MLSD	PMD [29]	CheckStyle [4]
Language support	Java, C# and others	Java	Java
Smells support	22	4	4
Dynamic threshold	Yes	No	No
Graphical support	Yes (Pie, Bar, Line Chart and 3-D)	Yes (data flow view)	Yes (Pie chart)
Plug-in of	Enterprise architecture	Eclipse	Eclipse
Refactoring	No	No	No
Code file access	Yes	Yes	Yes

Upon starting the tool, code smell definitions and detection algorithms are loaded and we get user interface as shown in Fig. 4. The user can select all code smells, types of code smells or a specific code smell from the menu.

We used the concept of dynamic threshold values for different metrics which are used for the detection of code smells. The dynamic thresholds give choice to the user to set threshold values at runtime for detection of different code smells. The results of MLSD by default are extracted in a tabular format. Each row in the table indicates the existence of a given smell. Each column contains a relevant fact (i.e., path in source code) regarding the given instance of a detected smell. Such detailed information is helpful for manual verification of detected code smells from the source code. One key feature of our tool is visualization of detected smells for better comprehension. The MLSD tool supports visualization of results in different 3D views such as column chart, pie chart, line chart, area chart, bubble chart. The visualization feature supports comprehension of detected smells from the large examined software projects. The user can also view the detected smells within the source code by clicking on recognized smells.

5.6 Limitations of Approach

To detect the code smells, our approach also depends on the metrics because the Data Model of Enterprise Archi-

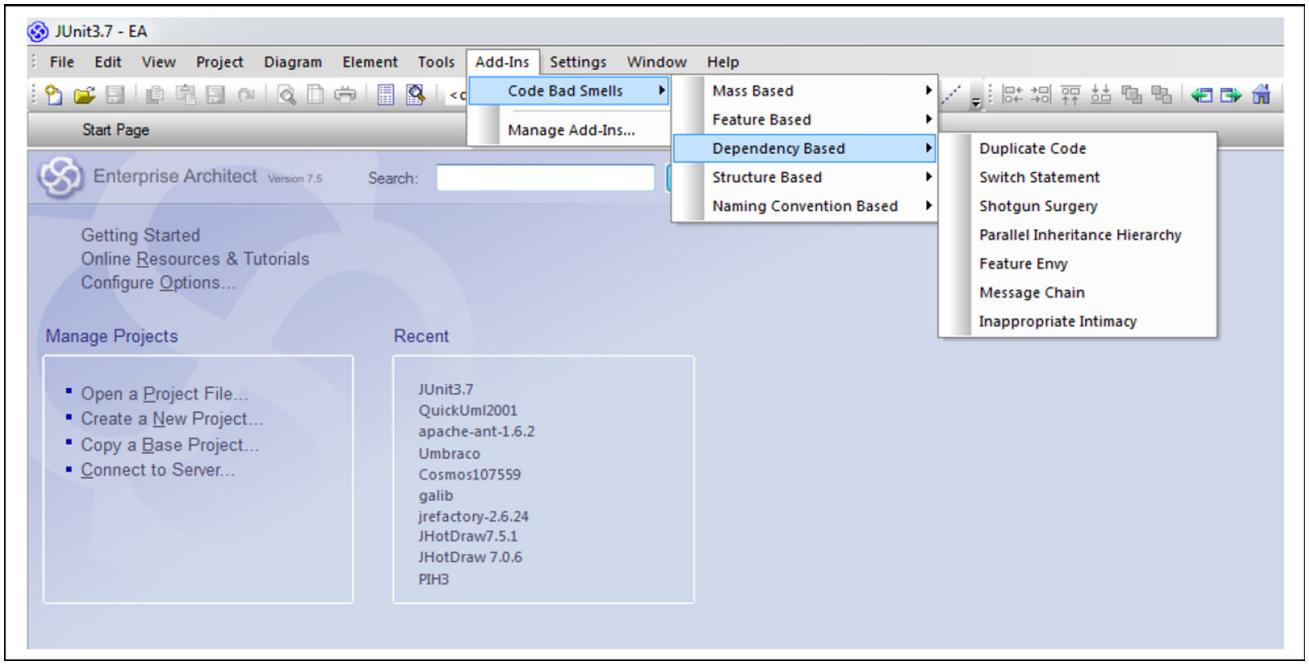

Fig. 4 Screenshot for user interface

tecture (although useful) is insufficient to provide all the required artifacts for the detection of code smells. To cope up with missing data, we directly analyze specific parts of source code using regular expressions. Regular expressions are a lightweight source code parsing approach, but they are syntax dependent, and we write different regular expressions to deal with each language. Sometimes, it becomes difficult to write generic regular expression patterns for matching nested information from the source code. Another limitation of our approach is that user must have knowledge about internal architecture of database model created through EA for extracting required information by using SQL and regular expressions. Lastly, we want to clarify that we did not compare our results of C# software systems with other tools due to unavailability of any free prototyping tool that can detect code smells from source code of C# language.

6 Evaluation of Approach

Evaluation of any approach is important to measure its quality, accuracy, performance and effectiveness. Initially, we performed experiments on single source code examples for the detection of code smells and validated our detection algorithms as a test case. The definitions of few code smells were refined after comparing our results manually with source code constructs. After initial experimentation, we evaluated our approach on eight open source software projects of Java and C# languages. The sizes of these software projects vary from small to large systems with code base ranging from 43 to 2135 classes which ensure scalability of our approach. The source code of all examined software projects is publicly available for comparison of results. The unique feature of our approach is support of multiple languages for code smells detection by using same detection algorithms only with updates of few regular expression patterns. Currently, we support Java and C# programming languages, but our approach is flexible for extension toward other programming languages. To the best of our knowledge, there is no publicly available code smells detection tool that supports detection of all 22 code smells from source code of C# programming language.

The fundamental information and statistics of examined software projects are listed in Table 5. The motivation for selection of these software projects is their free availability. The results extracted from these projects are presented in Tables 6 and 7. The data for each smell in Tables 6 and 7 represent smells extracted by our approach, number of true positives, number of false positives, number of false negatives, precision and recall.

Tables 6 and 7 show that our approach detected Long Parameter List, Long Method, Large Class, Lazy Class, Data Class, Speculative Generality, Switch Statement, Refuse Bequest and Parallel Inheritance Hierarchy code smells with 100 % accuracy. The accuracy of remaining code smells is also fair enough. The scalability of our approach is ensured by performing experiments on the eight open source software projects. The detailed results are available on the web for the readers [49].

Table 5 Statistics of examined software projects

Systems/features	Source	Language	Objective	Packages	Lines of code	Classes	Methods	Attributes
JUnit 3.7	www.junit.org	Java	Unit testing framework	9	9742	43	425	114
JHotDraw 7.0.6	www.jhotdraw.org	Java	Drawing editor	86	33,477	310	2562	420
QuickUML 2001	www.sourceforge.net/projects/quj/	Java	UML editor	13	46,572	204	1082	422
JRefactory 2.6.24	www.jrefactory.sourceforge.net	Java	Refactoring tool	58	216,244	562	4881	1367
RMS	www.f1sourcecode.com	C#	Restaurant Management System	60	13,833	78	653	434
CMS	www.f1sourcecode.com	C#	College Management System	64	79,339	267	3397	2393
Umbraco 6.1.1	https://umbraco.codeplex.com/	C#	CMS platform	444	173,537	1846	11,640	2577
Cosmos 107559	http://cosmos.codeplex.com/	C#	Operating system toolkit	355	296,394	2135	13,996	13,343

Table 6 Experimental results on 22 code smells for Java software projects

Smell	JUnit 3.7						QuickUML 2001						JRefactory 2.6.24						JHotDraw 7.0.6					
	SD	TP	FP	FN	P	R	SD	TP	FP	FN	P	R	SD	TP	FP	FN	P	R	SD	TP	FP	FN	P	R
1	2	2	0	0	1	1	5	5	0	0	1	1	2	2	0	0	1	1	11	11	0	0	1	1
2	1	1	0	0	1	1	0	0	0	0	1	1	2	2	0	0	1	1	2	2	0	0	1	1
3	0	0	0	0	1	1	0	0	0	0	1	1	4	4	0	0	1	1	1	1	0	0	1	1
4	15	15	0	0	1	1	81	81	0	0	1	1	159	159	0	0	1	1	87	87	0	0	1	1
5	0	0	0	0	1	1	0	0	0	0	1	1	2	2	0	0	1	1	0	0	0	0	1	1
6	0	0	0	0	1	1	2	2	0	0	1	1	2	2	0	0	1	1	4	4	0	0	1	1
7	0	0	0	0	1	1	3	3	0	0	1	1	82	82	0	0	1	1	23	23	0	0	1	1
8	0	0	0	0	1	1	35	24	11	0	.67	1	237	211	26	0	.89	1	43	40	3	0	.93	1
9	0	0	0	0	1	1	0	0	0	0	1	1	2	2	0	0	1	1	0	0	0	0	1	1
10	7	5	2	0	.71	1	120	97	23	0	.81	1	252	224	28	0	.90	1	215	192	23	0	.89	0
11	2	2	0	3	1	.60	19	14	5	13	.74	.52	20	18	2	3	.90	.86	25	20	5	12	.80	.62
12	11	10	1	0	.91	1	6	4	2	3	.67	.52	10	8	2	13	.80	.35	12	7	5	7	.58	.37
13	1	1	0	0	1	1	0	0	0	0	1	1	2	2	0	0	1	1	2	2	0	0	1	1
14	4	4	0	0	1	1	5	4	1	2	.80	.67	0	0	0	0	1	1	8	8	0	3	1	.73
15	0	0	0	0	1	1	0	0	0	0	1	1	0	0	0	0	1	1	0	0	0	0	1	1
16	6	5	1	1	.83	.83	59	52	7	9	.88	.85	252	249	3	18	.99	.93	288	271	17	40	.94	.87
17	33	30	3	0	.91	1	25	24	1	0	.96	1	136	122	14	0	.90	1	77	71	6	0	.92	1
18	0	0	0	0	1	1	0	0	0	0	1	1	2	2	0	0	1	1	0	0	0	0	1	1
19	12	12	0	0	1	1	54	54	0	0	1	1	110	110	0	0	1	1	76	76	0	0	1	1
20	0	0	0	0	1	1	1	1	0	0	1	1	2	2	0	0	1	1	0	0	0	0	1	1
21	2	1	1	4	.50	.20	5	1	4	6	.20	.14	23	18	5	0	.78	1	17	7	10	2	.41	.78
22	0	0	0	5	1	0	13	12	1	11	.92	.52	1	1	0	0	1	1	88	62	26	0	.70	1

1: Long Parameter List, 2: Long Method, 3: Large Class (God Class) 4: Lazy Class, 5: Data Class, 6: Speculative 7: Switch Statement, 8:Comment Class, 9: Incomplete Library Class, 10: Middle Man, 11: Divergent Changed, 12: Shotgun Surgery, 13: Message Chain, 14: Inappropriate Intimacy, 15: Refuse Bequest, 16: Feature Envy, 17: Temporary Field, 18: Alternative Classes with Different Interface, 19: Primitive Obsession, 20: Parallel Inheritance Hierarchy, 21: Data Clump, 22: Duplicate Code, SD: smells detected, TP: true positives, FP: false positives, FN: false negatives, P: precision, R: recall

Table 7 Experimental Results on 22 Code Smells for C# Software Projects

Smell	RMS						CMS						Umbraco 6.1.1						Cosmos 107559					
	SD	TP	FP	FN	P	R	SD	TP	FP	FN	P	R	SD	TP	FP	FN	P	R	SD	TP	FP	FN	P	R
1	18	18	0	0	1	1	75	75	0	0	1	1	43	43	0	0	1	1	119	119	0	0	1	1
2	12	12	0	0	1	1	57	57	0	0	1	1	7	7	0	0	1	1	53	53	0	0	1	1
3	1	1	0	0	1	1	8	8	0	0	1	1	5	5	0	0	1	1	2	2	0	0	1	1
4	9	9	0	0	1	1	81	81	0	0	1	1	559	559	0	0	1	1	755	755	0	0	1	1
5	0	0	0	0	1	1	29	29	0	0	1	1	114	114	0	0	1	1	44	44	0	0	1	1
6	0	0	0	0	1	1	0	0	0	0	1	1	21	21	0	0	1	1	44	44	0	0	1	1
7	0	0	0	0	1	1	0	0	0	0	1	1	0	0	0	0	1	1	1101	1101	0	0	1	1
8	19	14	5	0	.74	1	71	44	27	0	.62	1	404	278	126	0	.69	1	795	481	314	0	.60	1
9	0	0	0	0	1	1	0	0	0	0	1	1	0	0	0	0	1	1	0	0	0	0	1	1
10	0	0	0	0	1	1	58	34	24	18	.59	.85	868	631	237	114	.73	.85	0	0	0	132	1	0
11	0	0	0	2	1	0	0	0	0	7	1	0	28	21	7	26	.75	.45	38	29	9	25	.76	.54
12	0	0	0	3	1	0	0	0	0	11	1	0	23	11	12	27	.49	.29	31	9	22	48	.29	.16
13	0	0	0	3	1	0	3	1	2	18	.34	.05	5	2	3	23	.40	.08	19	4	15	30	.21	.12
14	0	0	0	1	1	0	0	0	0	13	1	.07	15	5	10	36	.33	.12	18	7	11	52	.39	.12
15	0	0	0	0	1	1	0	0	0	0	1	1	97	94	3	0	.97	1	207	207	0	0	1	1
16	0	0	0	0	1	1	0	0	0	0	1	1	716	666	50	160	.93	.81	45	41	4	35	.91	.54
17	65	60	5	0	1	1	434	388	46	0	.89	1	545	541	4	0	.99	1	767	687	80	0	.89	1
18	0	0	0	0	1	1	0	0	0	0	1	1	0	0	0	0	1	1	0	0	0	0	1	1
19	37	10	27	3	.57	.77	146	37	109	4	.60	.90	408	129	279	0	.32	1	1003	421	582	0	.42	1
20	0	0	0	0	1	1	0	0	0	0	1	1	0	0	0	0	1	1	113	113	0	0	1	1
21	3	3	4	3	.43	.50	32	23	9	7	.72	.92	18	12	6	15	.67	.44	112	79	33	19	.70	.81
22	4	2	2	1	.50	.67	0	0	0	0	1	1	0	0	0	45	1	0	0	0	0	102	1	0

Table 8 Precision, recall and *F* score of evaluated software projects

Project	TP	FP	FN	Precision (%)	Recall (%)	*F* score (%)
JUnit 3.7	88	8	13	91.6	87.1	88
QuickUML 2001	378	55	44	87.2	89.5	89
JRefactory 2.6.24	1222	80	34	93.8	97.3	97
JHotDraw 7.0.6	884	95	64	90.2	93.2	93
RMS	129	43	16	75.0	88.9	86
CMS	777	217	78	78.1	87.8	86
Umbraco 6.1.1	3139	737	446	80.9	87.5	86
Cosmos 107559	4196	1070	443	79.8	90.4	88

6.1 Evaluation of Accuracy

We measure accuracy of our approach using standard metrics such as precision, recall and *F* score. These metrics are generally used by all information retrieval techniques. Precision metric measures the correctness and recall measures the completeness of any approach. *F* score metric evaluates combined effect of both precision and recall. The formulas for these metrics are given below:

Precision = TP/(TP + FP), where TP: True Positives and FP: False Positives

Recall = TP/(TP + FN), where FN: False Negatives

F score = $((1 + w^2)(P*R))/(w^2P + R)$, where P = Precision and R = Recall

The recommend value of $w = 2.28$ as explained in [30]. Table 8 presents the evaluation of above metrics based on data from Table 5 on the examined software projects. The *F* score of our approach varies from 86 to 97 % for selected projects. We want to clarify that we apply one assumption in Tables 6 and 7 while calculating precision and recall. For example, when TP = 0, we assume that precision and recall are 1. The precision is calculated based on true positives and false positive instances of detected code smells for each software project. It was very laborious and time-consuming task to calculate recall for each examined software project due to unavailability of standard benchmark data. We analyzed source code of each project manually with the help of 8 master students that were studying a course of reverse engineering in COMSATS Institute of Information Technology, Lahore. We calculated accuracy of our approach based on true positives, false positives and false negatives instances of different smells. Each student was assigned one project for manual analysis of code smells from the source code against smells detected with our prototyping tool. Our results can be used by other researchers as benchmark for comparing results of code smell detection tools.

6.2 Evaluation of Performance

We evaluate the performance of our approach by executing our prototyping tool on the eight software projects while detecting all 22 code smells. The experiments are performed by using a desktop computer with Core i3 (dual-core) processor with 2 GB RAM. The runtime performance of our approach on four selected systems is shown in Fig. 5. Currently, our prototyping tool has performance issues for code smells Divergent Change, Message Chain, Shotgun surgery and Inappropriate Intimacy in the case of large software projects. These issues will be fixed by replacing regular expressions with source code parsers for extracting these four code smells. We want to clarify that we did not include the time that was used during reverse engineering of these systems in our evaluation. The Enterprise Architect modeling tool takes one and half minute to create intermediate representation of JUnit 3.7 project. This step is required just once, and then same models are used to extract smells from the recovered intermediate model which can be updated automatically after changes in the source code. The performance graph is extracted automatically by executing our prototyping tool on all code smells of four selected software projects.

6.3 Comparison and Discussion on Results

We compared our results with the previous code smells detection tools PMD and CheckStyle by running different open source examples. First reason for the selection of these tools is their availability. Secondly, these tools are also used by other researchers for comparing results of the same code smells [7]. Thirdly, these tools support detection of common subset of code smells. PMD scans Java source code and looks for potential problems or possible bugs like dead code, empty try/catch/finally/ switch statements, unused local variables or parameters, and duplicated code. PMD is able to detect Large Class, Long Method, Long Parameter List, and Duplicated code smells, and allows the user to set the threshold values for the exploited metrics [7]. CheckStyle has been developed to help programmers to write Java code that adheres to coding standards. It is able to detect Large Class, Long Method, Duplicated Code and Long Parameter List code smells [7]. The compared results are presented in Table 9. It is visible from results that there is a wide disparity in

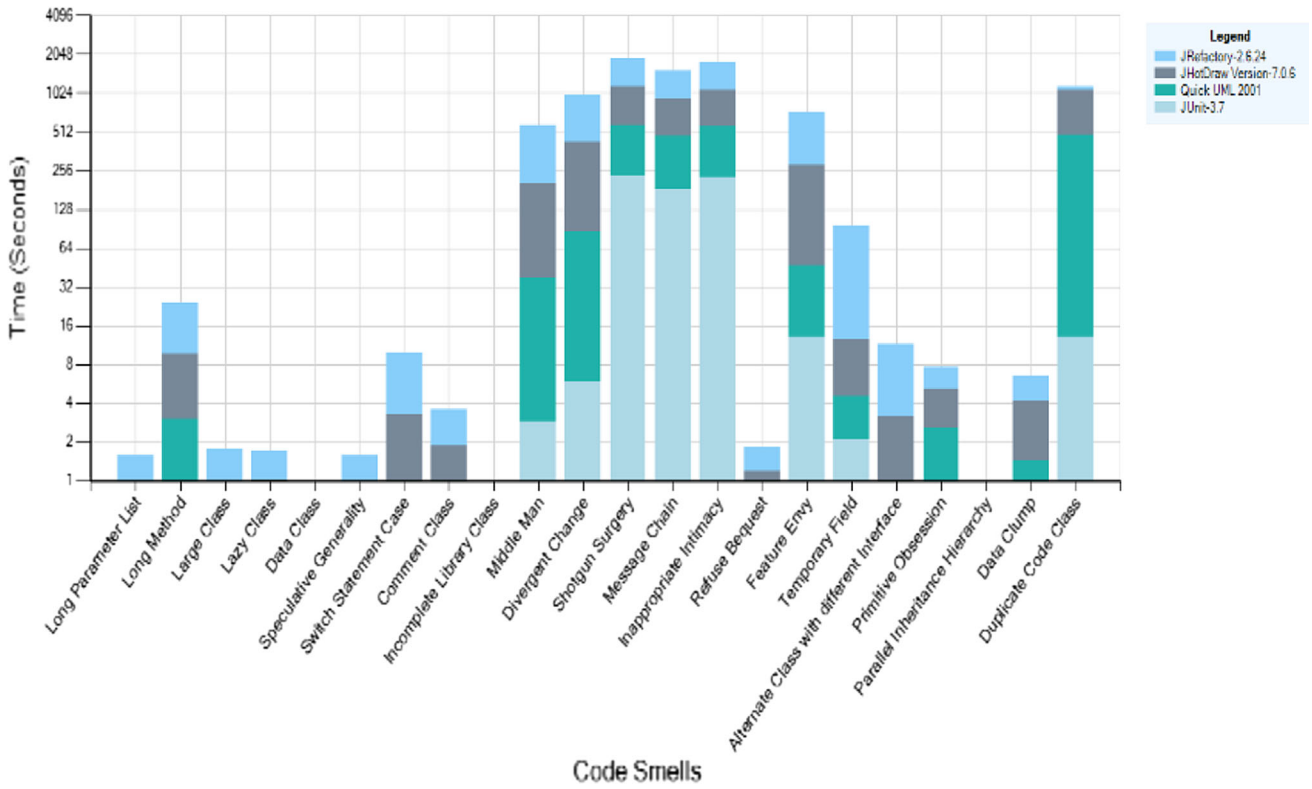

Fig. 5 Performance of prototype on four selected examples

Table 9 Comparison of our results with CheckStyle and PMD

Tools smells/ systems	Large class			Long method			Long parameter list			Large class			Long method			Long parameter list		
	ChS	MLSD	SS	ChS	MLSD	SS	ChS	MLSD	SS	PMD	MLSD	SS	PMD	MLSD	SS	ChS	MLSD	SS
JHotDraw Version 5.3	0	0	0	0	1	0	1	1	1	0	1	0	2	1	1	0	0	0
JHotDraw Version 7.0.6	1	1	1	3	2	2	4	14	4	1	3	1	14	8	8	0	0	0
JHotDraw Version 7.5.1	1	2	1	25	23	20	31	57	27	12	16	11	56	51	50	5	4	3
Quick UML 2001	0	0	0	0	0	0	3	6	3	0	0	0	2	0	0	1	1	1
JUNIT 3.7	0	0	0	1	1	1	2	2	2	0	0	0	1	1	1	0	0	0
Apache-Ant-1.6.2	3	4	3	20	18	16	11	16	10	15	30	12	68	58	55	1	1	1
JRefactory-2.6.24	5	4	3	5	2	2	0	2	0	11	11	10	10	4	4	0	0	0

ChS checkStyle, *SS* shared smells

the results of tools on the same examples while extracting code smells (Large Class, Long Method and Long Parameter List) from these selected examples. For example, CheckStyle, PMD and our prototyping tool extract 11, 1 and 16 Long Parameter List code smells from Apache-Ant 1.6.2 which clearly reflects disparity in results of these tools. It is also important to analyze which smells are commonly extracted by three applied tools. We went one step further and analyzed common smells extracted by these tools. This analysis is performed manually which is quite laborious and time-consuming task.

The common smells in Table 9 still do not reflect actual smells which are commonly extracted by these selected tools. We partially present common smells with actual smell name and its path in source code in Table 10. Disparities in the case of other smells are visible in Table 9. The major cause of sparse results is the use of different definitions and threshold values for detection of same code smells. In PMD tool, the default value of the threshold is 10 for Long Parameter List, while in CheckStyle it is 7 and some other tools use 3 threshold value for that smell. Large Class code smell is detected by most tools by counting Lines of code metric.

Table 10 Long Parameter List, large Class and Long Method shared smells

Smells	Software packages	Path in source code	Shared smells		
			ChS	PMD	MLSD
Long Parameter List	Junit 3.7	junit-3.7\src\awtui\TestRunner.java, Method: addGrid	✓	×	✓
		junit-3.7\src\swingui\TestRunner.java, Method: addGrid	✓	×	✓
	QuickUML 2001	QuickUml2001/Src/acme/GifEncoder.Java Method: GIFEncode	✓	✓	✓
		Jhotdraw7.5.1/src/main/java/org/jhotdraw/samples/svg/RadialGradient.Java Method: RadialGradient	✓	✓	✓
	JHotDraw 7.5.1	Jhotdraw7.5.1/src/main/java/org/jhotdraw/samples/svg/io/ SVGOutputFormat.Java Method: creaeRadialGradient	✓	✓	✓
	Apache-Ant-1.6.2	Jhotdraw7.5.1/src/main/java/org/jhotdraw/samples/svg/io/SVGOutputFormat.Java Method:createlinearGradient	✓	✓	✓
	JRefactory 2.6.24	Jhotdraw7.5.1/src/main/java/org/jhotdraw/samples/svg/io/SVGFigureFacory.Java Method: creaeRadialGradient	✓	✓	✓
		Jhotdraw7.5.1/src/main/java/org/jhotdraw/samples/svg/io/DefaultSVGFigureFacory.Java Method: creaeRadialGradient	✓	✓	✓
		Apache-Ant-1.6.2/src/main/org/apache/tools/ant/listener/MailLogger.Java Method: sendMimeMail	✓	✓	✓
		JRrefactory-2.6.24\src\org\acm\seguin\ide\JBuilder\TextStructureDelegate.java	✓	✓	✓
		Method:getTreeCellRendereComponent			
		JRrefactory-2.6.24\src\org\acm\seguin\parser\TokenMgrError.java Method: TokenMgrError	✓	✓	✓
Large Class	Junit 3.7	Jhotdraw7.5.1/src/main/java/org/jhotdraw/samples/svg/io/SVGInputFormat.Java	✓	✓	✓
	JRefactory 2.6.24	JHotDraw7.0.6/src/nanoxml/XMLElement.Java	✓	✓	✓
	JHotDraw 7.5.1	JRrefactory-2.6.24\src\org\acm\seguin\parser\JavaParser.java	✓	✓	✓
		JRefactory-2.6.24\src\org\acm\seguin\parser\JavaParserTokenManager.java	×	✓	✓
	JHotDraw 7.0.6	JRefactory-2.6.24\src\org\acm\seguin\summary\LineCountVisitor.java	×	✓	✓
	Apache-Ant- 1.6.2	JRefactory-2.6.24\src\org\acm\seguin\pretty\PrettyPrintVisitor.java	✓	✓	✓
		Apache-Ant-1.6.2/src/main/org/apache/tools/ant/Project.Java	✓	✓	✓
		Apache-Ant-1.6.2\org\apache\tools\Externalfiles\File.java	×	✓	✓
		Apache-Ant-1.6.2/src/main/org/apache/tools/ant/taskdefs/Javadoc.Java	✓	✓	✓
		Apache-Ant-1.6.2/src/main/org/apache/tools/ant/taskdefs/optional/net/FTP.Java	✓	✓	✓
Long Method	Junit 3.7	Class: junit-3.7\src\awtui\TestRunner.java	×	✓	✓
		Method:Protected Frame CreateUI			
	JRefactory 2.6.24	Class:JRefactory-2.6.24\src\ org\acm\seguin\tools\install\RefactoryInstaller.java	✓	✓	✓

Table 10 continued

Smells	Software packages	Path in source code	Shared smells		
			ChS	PMD	MLSD
	JHotDraw 7.0.6	Method: Private void prettySettings			
	Apache-Ant- 1.6.2	Class: JRefactory-2.6.24\src\ org\acm\seguin\tools\install\RefactoryInstaller.java	×	✓	✓
		Method: Public void run			
		Class: JHotDraw7.0.6\src\org\jhotdraw\geom\DoubleStroke.java	×	✓	✓
		Method: Protected void traceStroke			
		Class: JHotDraw 7.0.6\src\net\n3\nanoxml\StdXMLParser.javaProtected	×	✓	✓
		Method: void processElement			
		Class:Apache-Ant-1.6.2\src\testcases\org\apache\tools\ant\taskdefs\optional\i18n\IntrospectionHelperTest.java	✓	×	✓
		Method: void testAttributeSetters			
		Class:Apache-Ant-1.6.2\org\apache\tools\ant\taskdefs\optional\ejb\JonasDeploymentTool.java	×	✓	✓
		Method: Private void addGenICGeneratedFiles			

✓ = Yes, × = no

The threshold values for the detection of Large Class code smell are 1000 and 2000 by PMD and CheckStyle tools, respectively. Furthermore, some code smells detection tools also use cyclomatic complexity metric with Lines of code metrics for the detection of Large Class. Similarly, threshold value for Long Method code smell is 100 and 150 for PMD and CheckStyle tools, respectively. The variations in the definitions and threshold values cause disparity in the results of these tools for the detection of same code smells. We provide an option of dynamic threshold values in our prototyping tool for comparison of our results with PMD and CheckStyle. We compared only results of these three smells with PMD and CheckStyle tools as they are commonly extracted smells by these tools.

6.4 Benchmark of Results

Benchmarks are very important for comparing, analyzing and validating results of existing and new code smell detection approaches and tools. We cannot find any standard benchmark system for results of code smell detection tools. Establishing benchmark systems for large software projects is a daunting and time-consuming task. A number of code smells detection tools are available, but their results are diverse on the same examined systems. The benchmarks for few code smells are available, but we cannot find a complete and acceptable benchmark system that contains results of all 22 code smells. We publish results of our tool on the eight open source systems as an initial attempt toward a benchmark system for researchers working in the field of code smells detection. We provide detailed information about detected smells including file path in source code, important parameters related to each smell, threshold values. We plan to extend and make our benchmark interactive to get feedback from researchers in the future. The results of other code smells detection tools will also be uploaded on the benchmark for the evaluation and comparisons. The results of benchmark are available on the web source [49]. We request feedback from community on the initial version of presented benchmark for updates and improvements.

6.5 Threats to Validity

Validity is a major requirement for empirically validating results of different approaches for researchers and practitioners [51]. External validity threat refers to generalization of results for large data sets. Our approach mitigates that threat as we performed experiments on the eight open source systems of different sizes for evaluation of our approach. However, our approach is not tested on industrial applications due to unavailability of source code for these applications. Our experiments also focus on all code

smells as compared with previous approaches which are restricted to only a few code smells. For internal validity, the lack of standard definitions for code smells and unavailability of standard benchmark systems for results of code smell detection techniques is a major concern. Our proposed approach mitigates threat of varying code smells definitions by providing customizable algorithms and threshold values of metrics. Second, we compare results of our approach with existing approaches on smell by smell basis for validity of our results. Regarding reliability validity of our approach and results, the selected examples are all open source software projects and source code is available on the web for validation. The list of smells definitions, detection algorithms and results of tool are also available on the web which eliminates reliability threats. However, there might be reliability threats while checking our results for the calculation of recall in the case of large software projects.

7 Conclusion and Future Work

The recovery of code smells from source code supports code refactoring, program comprehension, maintenance and highlights poor practices adopted by developers in the source code. A number of code smell recovery approaches and tools are presented by different authors, but they still suffer accuracy and flexibility issues. Our proposed approach aims at addressing these issues based on five major contributions. First, we propose a new taxonomy that classifies all 22 code smells into five groups for their better understanding. Second, we present customizable definitions and algorithms for detection of code smells from multiple languages with varying features. Thirdly, we apply SQL queries and regular expressions for matching definitions of code smells in the source code and these searching queries are not hard coded in the source code. Our approach is capable of detecting and visualizing all 22 code smells from source code of Java and C# languages. Fourthly, a prototyping tool is developed to validate the concept of approach. We evaluate our tool on eight open source software projects implemented in two programming languages and recover all code smells with improved accuracy. The results of presented approach are compared with state-of-the-art approaches. Our results illustrate the significance of customizable code smell definitions and lightweight searching techniques in order to overcome the accuracy and flexibility issues of previous approaches. Finally, a benchmark system for results of code smells is published as an initial attempt. We plan to extend our approach for refactoring of recovered code smells. The future work will also focus on detection of other code smells, design smells and antipatterns.

Acknowledgments Authors acknowledge the valuable feedback of reviewers on the early version of this paper.

Appendix: Code Smell Definitions

Names	Definitions	Specifications
Long Parameter List	A Long Parameter List exists if	
	1. A method contains more than or equal to a specified number of parameters. The specific (threshold) value for the number of parameters is given by the user or	PL ≥ PLV OR RDT ≥ 1
	2. It contains two or more (specified numbers of) parameter of same primitive data type.	Where PLV = Value for Maximum Number of Parameters, RDT = Replicate Primitive Data Type.
Long Method	A long function exists if a function contains	
	1. More than or equal to a specified number of Lines of code (excluding method declaration, comments and parenthesis) or	LOCM ≥ LOCMV
	2. More than or equal to a specified number of Operands and (or) Operators;	Where LOCM = Lines of Code in a Method, LOCMV = Value for total LOC in a method
Large Class (God Class)	A Large or God Class exists if a class contains	
	1. More than or equal to a specified number of Lines of	LOCC ≥ LOCCV OR nMC ≥ nMCV
	code (excluding method declaration, comments and parenthesis) or (and); (PMD uses 100 as specified threshold while for Check Style it is 150)	Where LOCC = Lines of Code in a class, LOCCV = Value for total LOC in a class, nMC = Number of methods in a class, nMCV = Value for total number of methods in a class
	2. More than or equal to a specified number of global variables and methods.	
Lazy Class	A Lazy Class exists if a class contains	
	1. Less than or equal to a specified number of lines of code (excluding method declaration, comments and parenthesis) or (and);	LOCC ≥ LOCCV OR nMC ≥ nMCV
	2. Less than or equal to a specified number of methods [2]	Where LOCC = Lines of code in a class, LOCCV = Value for total LOC in a class, nMC = Number of methods in a class, nMCV = Value for total number of methods in a class
Data Class	A Data Class exists if a class contains	
	1. Only setters, getters or (and) constructor but no other method to perform any functionality [2];	nMc(nf) ≥ nMc(nf)V OR nMc = 0
		Where nMc(nf) = Number of non-functional methods in a class, nMc(nf)V = Specified value for total number of non-functional methods in a class
Speculative Generality (dead code)	Speculative Generality exists if	
	1. There is an abstract class which is not implemented anywhere in the system [45]	UAC ≥ 1
		Where UAC = Un-implemented abstract classes
Middle Man	Middle Man smell exists if a class contains	
	1. More than or equal to a specified percentage of external method calls to the total numbers of methods it contains [2]	xMC/nMC ≥ specified%
		Where xMC = external Method Call, nMC = Number of Methods in a class
Divergent Changed	A class C is exposed to Divergent Changed if it contains	
	1. More than or equal to a specified number of distinct external methods called by C or [46];	xMC ≥ xMCv OR xCC ≥ xCCv
	2. More than or equal to a specified number of distinct classes whose method(s) are called from C [46]	Method call value, xCC = External class call, xCCv = External class call value

Names	Definitions	Specifications
Duplicate Code	Duplicate Code class exist if 1. Two or more classes contains more than or equal to a specified percentage of similar code;	LOCC1(c2) (cmn) / LOCC1(c2) (total) $\geq$ % Where LOCC1(c2) (cmn) = Lines of common code in Class C1 and C2, LOCC1(c2) (total) = Total lines of code in class C1 and C2
Switch Statement	A Switch Statement in a code is a code smell if it contains: 1. More than or equal to a specified number of case statements in it [56].	SSC $\geq$ SSCV Where SSC = Switch Statement Case, SSCV = specified value for total number of Switch Statement Cases.
Shotgun surgery	A method M is exposed to Shotgun surgery if an application contains: 1. More than or equal to a specified number of distinct external methods that calls M or [46]; 2. More than or equal to a specified number of distinct classes that calls M [46]	xMCM $\geq$ nxMCM OR xCCM geq nxCCM Where xMCM = External Method Call for Method M, nxMCM = number of External Method Calls for Method M. xCCM = External Class Call for Method M, nxCCM = number of External Class Call for Method M
Parallel Inheritance Hierarchy	Class x and Class y are in Parallel Inheritance Hierarchy Smell if: 1. They are at least at second or greater level of hierarchy (having two or more parents at their upper levels of hierarchy) and; 2. Both x and y are at same level of hierarchy and; 3. Both x and y have same number of children [45]	PIHxy = DIT x, $y > 2$ + DITx = DITy + NOCx = NOCy Where PIH = Parallel Inheritance Hierarchy between class x and y, DIT = Depth inheritance hierarchy, NOC = Number of Children
Feature Envy	Feature Envy smell exists if a method contains 1. More than or equal to a specified number of external method calls [2];	xMC $\geq$ xMCv Where xMC = External method call, xMCv = External method call value
Message Chain	Message Chain smell exists if 1. Method calling chain in the system becomes equal to or more than a specified value [45];	MC$\rightarrow\rightarrow$ $\geq$ MC$\rightarrow\rightarrow$V Where MC$\rightarrow\rightarrow$ = Method calling chain, MC$\rightarrow\rightarrow$V = Method calling chain value
Inappropriate intimacy	Inappropriate intimacy exists at 1. Class level if a calling class C1 calls more than or equal to a specified number of distinct methods of class C2 or; 2, method level if a method M of class C1 calls more than or equal to a specified number of times the distinct methods of class C2;	MCC2(C1) $\geq$ nMiC OR MCC2(C1M) $\geq$ nMiC Where MCC2(C1) = Number of Methods of class C2 called by class C1 nMiC= Number of intimated Methods in a class MCC2(C1M) = Methods of Class C2 called by method M of class C1
Temporary Field	A field F is a Temporary Field if a class contains 1. Less than or equal to a specified percentage of methods that uses F to the total numbers of methods it contains [26];	Mf/nMc <= specified% Where Mf = Method that uses field F nMc= Number of methods of a class

Names	Definitions	Specifications
Refuse Bequest	Refuse Bequest exists if a Child Class Cc 1. Only uses less than or equal to a specified percentage of protected methods of its parent class Cp [44]	(MCc (proc)/MCp (proc))*100 <= specified% Where MCc (proc) = Protected methods of parent class used in Child Class Cc, MCp (proc) = Total protected methods in parent class Cp
Data Clump	Data Clump exists if 1. Two or more classes have two or more variables of same name and type.	Cn(Vn) ∈ SVN And SDT Similar name and type Where Cn(Vn)= Two or more classes with n number of variables SVN= Same variable names, SDT= Same data type
Primitive obsession	Primitive obsession exists if a class contains 1. More than or equal to a specified number of primitive data types [45];	nPVC ≥ PVcV Where nPVC = number of primitive variables in a class, PVcV = Value for primitive variables in a class
Comments	Comments are considered as code smell if a class contains 1. More than or equal to specified percentage of commented code (as compared) to its total lines of code where commented code includes inline comments, single-line comments and multi-line comments	LOCcmntd/LOCTotal ≥ specified% Where LOCcmntd = Commented line of code in a class, LOCTotal = Total line of code in a class
Alternative Classes with Different Interface	Alternative Classes with Different Interfaces occurs if class C1 and C2 has similarity function. Similarity function includes if classes has 1. Same number of methods and; 2. More than or equal to specified percentage of code similar to one other	SF c1, c2 ≥ specified% Where SF c1, c2 = Similarity Factor of Class 1 and Class 2

References

1. Abbes, M.; Khomh, F.; Guéhéneuc, Y.G.; Antoniol, G.: An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In: Proceedings of 15th European Conference on Software Maintenance and Reengineering, pp. 181–190 (2011)
2. Arcelli, F.; Tosi, C.; Zanoni, M.; Maggioni, S.: The MARPLE project: A tool for design pattern detection and software architecture reconstruction. In: 1st International Workshop on Academic Software Development Tools and Techniques (WASDeTT-1) (2008)
3. Chaikalis, T.; Tsantalis, N.; Chatzigeorgiou, A.: JDeodorant: identification and removal of type-checking bad smells. In: Proceeding of 12th European Conference on Software Maintenance and Reengineering, pp. 329–331 (2008)
4. CheckStyle. http://checkstyle.sourceforge.net
5. Dehagani, S.M.H.; Hajrahimi, N.: Which factors affect software projects maintenance cost more? Acta Inform. Med. **21**(1), 63 (2013)
6. Fard, A.M.; Mesbah, A.: Jsnose: detecting javascript code smells. In: Proceedings of 13th International Working Conference on Source Code Analysis and Manipulation, pp. 116–125 (2013)
7. Fontana, F.A.; Braionea, P.; Zanonia, M.: Automatic detection of bad smells in code: an experimental assessment. J. Object Technol. **11**(2), 1–38 (2012)
8. Fowler, M.; Beck, K.; Brant, J.; Opdyke, W.; Robert, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Reading (1999)
9. Kiefer C.; Bernstein A.; Tappolet J.: Mining software repositories with iSPAROL and a software evolution ontology. In: Fourth International Workshop on Mining Software Repositories (MSR), p. 10 (2007)
10. Lozano, A.; Wermelinger, M.; Nuseibeh, B.: Assessing the impact of bad smells using historical information. In: Ninth International Workshop on Principles of Software Evolution: in Conjunction with the 6th ESEC/FSE Joint Meeting, pp. 31–34 (2007)
11. Mäntylä, M.V.; Lassenius, C.: Subjective evaluation of software evolvability using code smells: an empirical study. J. Empir. Softw. Eng. **11**(3), 395–431 (2006)
12. Mantyla, M.; Vanhanen J.; Lassenius C.: Bad smells-humans as code critics. In: Proceedings of the International Conference on Software Maintenance, pp. 399–408, (2004)
13. Marinescu, R.: Detection strategies: metrics-based rules for detecting design flaws. In: Proceedings of 20th International Conference on Software Maintenance, pp. 350–359 (2004)
14. Moha, N.; Guéhéneuc, Y.; Duchien, L.; Meur, A.L.: DECOR: a method for the specification and detection of code and design smells. IEEE Trans. Softw. Eng. **36**(1), 20–36 (2010)
15. Oliveira, P.; Valente, M.T.; Paim Lima, F.: Extracting relative thresholds for source code metrics. In: Software Evolution Week: IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE'2014), pp. 254–263. Antwerp, Belgium: IEEE Computer Society (2014)
16. Palomba F.; Bavota G.; Penta MD.; Oliveto R.; De Lucia A.; Poshyvanyk D.: Detecting bad smells in source code using change history information. In: Proceedings of 28th IEEE/ACM International

Conference on Automated Software Engineering, pp. 268–278 (2013)
17. Palomba, F.; Bavota, G.; Di Penta, M.; Oliveto, R.; Poshyvanyk, D.; De Lucia, A.: Mining version histories for detecting code smells. IEEE Trans. Softw. Eng. **41**(5), 462–489 (2015)
18. Rasool, G.; Arshad, Z.: A review of code smell mining techniques. J. Softw.: Evol. Process **27**(11), 867–895 (2015)
19. Ratiu D.; Ducasse S.; Gîrba T.; Marinescu R.: Using history information to improve design flaws detection. In: Proceedings of the Eight Euromicro Working Conference on Software Maintenance and Reengineering, p. 223 (2004)
20. Sjoberg, D.I.K.; Yamashita, A.; Anda, B.C.D.; Mockus, A.; Dyba, T.: Quantifying the effect of code smells on maintenance effort. IEEE Trans. Softw. Eng. **39**(8), 1144–1156 (2013)
21. Sparx System Enterprise Architect Modeling tool, homepage: http://www.sparxsystems.com/
22. Marcus, A.; Maletic, J.: Identification of high-level concept clones in source code. In: Proceedings of 16th Annual International Conference on Automated Software Engineering, pp. 107–114 (2001)
23. Marinescu, R.; Ratiu, D.: Quantifying the quality of object-oriented design: the factor-strategy model. In: Proceedings of Working Conference on Reverse Engineering, pp. 192–201 (2004)
24. Marticorena, R.; Crespo, Y.: Parallel inheritance hierarchy: detection from a static view of the system. In: 6th International Workshop on Object Oriented Reengineering, p. 6 (2005)
25. Mihancea P.F.; Marinescu R.: Towards the optimization of automatic detection of design flaws in object-oriented software systems. In: Proceedings of Ninth European Conference on Software Maintenance and Reengineering, pp. 92–101 (2005)
26. Munro, M.J.: Product metrics for automatic identification of bad smell design problems in java source-code. In: 11th IEEE International Symposium on Software Metrics, p. 15 (2005)
27. Naveen R.: JSmell: a bad smell detection tool for java systems. Dissertation and theses (2009)
28. Nitin M.: JAVA SMELL DETECTOR, Master thesis, San Jose State University (2011)
29. PMD. http://pmd.sourceforge.net
30. Pettersson, N.; Löwe, W.; Nivre, J.: Evaluation of accuracy in design pattern occurrence detection. Proc. IEEE Trans. Softw. Eng. **36**(4), 575–590 (2010)
31. Rasool, G.; Asif, N.: Software artifacts recovery using abstract regular expressions. In: Proceedings of International Multitopic Conference (INMIC'2007), pp. 1–6 (2007)
32. Rasool, G.; Philippow, I.; Mäeder, P.: Design pattern recovery based on annotations. Adv. Eng. Softw. **41**(4), 519–526 (2010)
33. Nongpong K.: Integrating Code Smells Detection with Refactoring Tool Support. Ph.D thesis, The University of Wisconsin-Milwaukee (2012)
34. Gopalan R.: Automatic detection of code smells in java source code. Dissertation for Honour Degree, The University of Western Australia (2012)
35. Slinger, S.: Code smell detection in eclipse. Thesis report, Delft University of Technology (2005)
36. Sahin, D.; Kessentini, M.; Bechikh, S.; Deb, K.: Code-smells detection as a bi-level problem. ACM Trans. Softw. Eng. Methodol. **24**(1), 1–44 (2014)
37. Kim, T.W.; Kim, T.G.; Seu, J.H.: Specification and automated detection of code smells using ocl. Int. J. Softw. Eng. Appl. **7**(4), 35–44 (2013)
38. Kessentini, W.; Kessentini, M.; Sahraoui, H.; Bechikh, S.; Ouni, A.: A cooperative parallel search-based software engineering approach for code-smells detection. IEEE Trans. Softw. Eng. **40**(9), 841–861 (2014)
39. dos Santos Neto, B.F.; Ribeiro, M.; da Silva, V.T.; Braga, C.; de Lucena, C.J.P.; de Barros Costa, E.: AutoRefactoring: a platform to build refactoring agents. Expert Syst. Appl. **42**(3), 1652–1664 (2015)
40. Ganea, G.; Verebi, I.; Marinescu, R.: Continuous quality assessment with inCode. Sci. Comput. Program. 1–18 (2015). doi:10.1016/j.scico.2015.02.007
41. Ierusalimschy, R.: A text pattern matching tool based on parsing expression grammars. Softw.: Pract. Exp. **39**(3), 221–258 (2009)
42. InFusion Code Smells tool. http://www.intooitus.com/products/infusion [Accessed on 15-01-2016]
43. Khamis, N.; Witte, R.; Rilling, J.: Automatic quality assessment of source code comments: the JavadocMiner. In: Hopfe, C.J., Rezgui, Y., Métais, E., Preece, A., Li, H. (eds.) Natural Language Processing and Information Systems. Lecture Notes in Computer Science, vol. 6177, pp. 68–79. Springer, Berlin, Heidelberg (2010)
44. Lanza, M.; Marinescu, R.; Ducasse, S.: Object-oriented metrics in practice. pp. 29–31. Springer, Heidelberg (2006)
45. Roperia, N.: JSmell: A Bad Smell Detection Tool for Java Systems. California State University, Long Beach (2009)
46. Macia, I.; Garcia, J.; Popescu, D.; Garcia, A.; Medvidovic, N.; von Staa, A.: Are automatically-detected code anomalies relevant to architectural modularity?: an exploratory analysis of evolving systems, In: Proceedings of the 11th Annual International conference on Aspect-Oriented Software Development, pp. 167–178. ACM, Potsdam, Germany (2012)
47. Mantyla M.; Vanhanen J.; Lassenius C.: Bad smells in software-a taxonomy and an empirical study. In: Proceedings of ICSM, pp. 381–384 (2003)
48. Mens T.; Tourwe T.; Munoz F.: Beyond the refactoring browser: advanced tool support for software refactoring. In: Proceedings of the Sixth International Workshop on Principles of Software Evolution, pp. 39–44 (2003)
49. Web link for results: Experimental results for the eight evaluated systems, 2015. URL http://research.ciitlahore.edu.pk/Groups/SERC/CodeSmells.aspx
50. Rasool, G.; Mäder, P.: Flexible design pattern detection based on feature types. In: Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 243–252 (2011)
51. Wohlin, C.; Runeson, P.; Höst, M.; Ohlsson, M.C.; Regnell, B.; Wesslén, A.: Experimentation in Software Engineering. Springer, Berlin (2012)
52. Tsantalis, N.; Chatzigeorgiou, A.: Identification of move method refactoring opportunities. IEEE Trans. Softw. Eng. **35**(3), 347–367 (2009)
53. Yamashita, A.; Counsell, S.: Code smells as system-level indicators of maintainability: an empirical study. J. Syst. Softw. **86**(10), 2639–2653 (2013)
54. Yamashita A.; Moonen L.: Exploring the impact of inter-smells relationships in the maintainability of a system: an empirical study. In: Proceedings of International Conference on Software Engineering, pp. 682–691 (2013)
55. Zhang, M.; Hall, T.; Baddoo, N.: Code bad smells: a review of current knowledge. J. Softw. Maint. Evol.: Res. Pract. **23**(3), 179–202 (2011)
56. Zhang, M.; Hall, T.; Baddoo, N.; Wernick, P.: Do bad smells indicate trouble in code?. In: Proceedings of the 2008 Workshop on Defects in Large Software Systems, pp. 43–44 (2008)