Journal of Scientific & Industrial Research
Vol. 85, July 2026, pp. 590-601
DOI: 10.56042/jsir.v85i7.17036

Detection of Shotgun Surgery Code Smell using Structural and Evolutionary Features of Source Code

Kapil Sharma* & Jitender Kumar Chhabra

Computer Engineering Department, National Institute of Technology, Kurukshetra 136 119, Haryana, INDIA

Received 10 February 2025; revised 16 July 2026; accepted 22 October 2026

"Shotgun surgery" is a code smell in which a single change in a software artifact is likely to initiate several other changes across multiple software classes, thereby decreasing software design quality. This deficient design leads to technical debt, which may increase the likelihood of future bugs because the design shortcuts, code complexity, and degraded-maintainability make subsequent changes more error-prone. The existing approaches of smell detection are dependent on one of the two: structural dependencies derived from the source code or evolutionary dependencies derived from the version control system. Ideally both dependencies are desirable for change propagation, as code smell is not a static phenomenon. Hence, this study proposes an integrated approach for code smell detection by combining information from structural and evolutionary software sources, both being computed in a new and robust manner. Structural dependencies among classes are measured in the form of a Structural Dependency Score (SDS), considering multiple class dependencies. Evolutionary dependencies are measured through a Change Impact Score (CIS) considering change frequency, code churn information, developer overlap, and temporal information. The weighted integration of these two dependencies' scores is used to derive a Total Dependency Score (TDS), which is further used to generate the direct and indirect dependency adjacency matrices between each pair of classes. Then these matrices are used to compute propagation and impact scores, both of which are subsequently combined to calculate the smell intensity, which helps in the smell classification. This approach is empirically evaluated on four open-source Java software systems and compared with existing approaches individually as structural and evolutionary baselines. The results indicate that the proposed methodology provides a better and more accurate mechanism to detect Shotgun Surgery.

Keywords: Change impact score, Design quality, Evolutionary information, Software maintenance, Structural information

Introduction

Code smells are indicators of poor source code design. Fowler and Beck introduced this word to describe deviations from object-oriented design principles and proposed 22 types of code smells, including Shotgun Surgery, Long Method, and God Class.[1] Code smells, also referred to as antipatterns[2–4], do not necessarily show the presence of bugs but degrade software quality and maintainability, which makes code comprehension very difficult, and accumulate technical debt. Their detection and timely removal by applying refactoring improves the code structure without changing the external behavior and can solve these problems.

Code smells have been classified into bloater, object-oriented abuser, dispensable, and change-preventer based on their unique characteristics.[5] When the size of the codebase surpasses a threshold, which leads to maintenance challenges, this situation has been classified as a "bloater." The code smell is classified as an object-oriented abuser when the design of the source code demonstrates excessive coupling between artifacts, which is not desirable. When the source code has unnecessary elements, this category is classified as dispensable, and when the design of the source code prevents the addition of any new features, this situation is categorized as a change-preventer category. This category encompasses smells that lead to increased code maintenance, often referred to as maintenance-related smells.[6] Shotgun Surgery code smell is a maintenance-related code smell.

The presence of the shotgun surgery smell initiates changes across multiple classes due to a change in a single class. These scattered changes lead to degraded software design and reduced maintainability because developers must perform the detection and understanding as well as modification of multiple software artifacts in order to complete a single change. The existing method for this smell detection

*Author for Correspondence
E-mail: kskapilsharma777@gmail.com

depends primarily on structural information or evolutionary information in isolation, thereby capturing only partial information related to scattered-change behavior. To solve this research gap, this study proposes an integrated smell detection approach that combines structural information through structural dependencies extracted from source code with evolutionary information through evolutionary dependencies derived from software change history. Both types of dependencies are quantified separately and subsequently weighted and integrated for shotgun surgery detection. This empirical evaluation is limited to only Java-based object-oriented systems, although this framework can be extended to other object-oriented programming languages also with the availability of appropriate parsing and dependency-extraction software tools. The major contributions of this paper are:

- A new approach for Shotgun Surgery code smell detection has been proposed with a combination of the structural and evolutionary information of the source code.
- This approach introduces new evolutionary metrics computed in more robust manner to measure the change history information of the source code along with both direct and indirect structural dependencies among classes.
- To examine its effectiveness, the proposed approach has been compared with the existing smell detection approaches, and also analyzed in comparison with structural and evolutionary features.
- The proposed technique has been empirically evaluated on four open-source Java projects from different domains, and 33.33% of the smelly candidates detected across these projects were associated with buggy classes, indicating the maintenance relevance of this integrated smell detection method.

Related Work

Software code smell detection is an important area in the software engineering research domain due to its impact on software design quality and maintainability. Shotgun Surgery is a maintenance-related code smell where a single conceptual change initiates small modifications across multiple software classes. These scattered changes increase maintenance effort and may produce inconsistencies and unintended faults in software evolution, leading to the need for effective Shotgun Surgery code smell detection. This section reviews several code smell detection methods, with a particular focus on the Shotgun Surgery smell. Code smell detection is a subjective phenomenon, as multiple smells are characterized by design symptoms based on developer perception rather than on universally accepted rules and heuristics. Consequently, smell detection methods differ in their selection of related software metrics, their thresholds, their detection rules, and their analysis granularity, whether at the class level or method level. These differences may result in inconsistent results for smell detection, even when applied to the same software project. This difference primarily arises due to the smell interpretation and quantification by the developer based on their perception rather than the availability of multiple detection approaches.

Existing approaches for code smell detection include manual-based, metrics-based, visualization-based, and automated methods. The manual detection method needs domain knowledge to interpret project-specific software design characteristics that automated software tools may have missed out on. However, this approach is time-consuming, resource intensive, and prone to human error.[7,8] Metrics-based smell detection methods quantify software design characteristics using the established Chidamber and Kemerer Metrics Suite[9] and McCabe Metrics Suite[10],although inappropriate threshold values of used software metrics may result in false positives.[11] Lanza and Marinescu used combinations of different software metrics and found thresholds to detect design disharmonies.[12] Visualization-based smell detection approaches exhibit software characteristics, including size, complexity, coupling, and cohesion, through pictorial representations to find software classes that exhibit unusual structural behavior. However, the actual smell detection result using this method primarily depends on additional criteria, such as relevant metrics selection, threshold tuning, detection rules, or developer assessment.[13,14] Automated tools for analyzing software quality, including PMD[15], Checkstyle[16], and JDeodorant[17], used predefined rules and heuristics to verify code quality and detect potential refactoring candidates. Although these tools are used for analyzing large software systems, they do not provide code semantics information. Beyond these approaches, researchers have also explored machine learning, deep learning, dynamic analysis, and change history mining for code smell detection.

Machine learning has significantly advanced the code smell detection. Fontana *et al*. proposed an approach and developed a classifier using a decision tree and random forest for God class and Feature envy detection. They exploited machine learning with the code metrics as input features and ranked smelly instances based on their severity level.[18] Fontana *et al*. also performed a comparative analysis with various machine learning techniques for code smell detection in another study.[19] Tsantalis *et al.* used structural and behavioral data from methods to detect type-checking code smell and replace smelly instances with refactoring, like replacing type code with state/strategy.[20] Dewangan *et al.* proposed an approach using two deep learning models and five ensemble machine learning models for four types of code smell detection; for dataset balancing, they used SMOTE, and for relevant feature selection, they used the chi-square test, with the highest accuracy of 100% measured for Long method and 91.45% calculated for Feature envy.[21] Deep learning has also been used for smell detection, Liu *et al.* used a deep neural network to select smell relevant features and mapped these features for smell detection.[22] Another approach, *Dele Smell*, proposed by Zhang *et al.*, where the accuracy of deep learning models increased by balancing the dataset using the refactoring tool, generates positive-smelling instances automatically and uses both semantic and structural features through the latent semantic analysis approach. The detected smells are only Brain method and Brain classes.[23]

Code smells evolve with software changes; hence, evolutionary information is valuable for smell detection. Khomh *et al*. investigated the code change history to detect smelly candidates using file co-change patterns.[24] More than evolutionary information, Hozano *et al*. proposed an approach using developer-feedback to improve software design quality for four code smells.[25] Saranya *et al*. proposed a metaheuristic-based smell detection approach using Hybrid Particle Swarm Optimization (HPSOM) by evaluating nine open-source projects. Their results reported 94% precision and 92% recall in the detection of Blob, Spaghetti code, Feature Envy, and Functional decomposition smells.[26]

The Shotgun Surgery code smell is characterized by the need for scattered changes across multiple software artifacts to implement a single conceptual change. Multiple studies have performed its detection using evolutionary information derived through software change history. Fu and Shen suggested an association-rule-mining approach that leverages software change history to detect code smells like Duplicated Code, Shotgun Surgery, and Divergent Change.[27] Palomba *et al.* proposed a Historical Information for Smell Detection (HIST), which uses change history to detect maintenance-related smells, like Divergent change Parallel inheritance, along with Shotgun Surgery code smell.[6] Although these methods highlight the role of evolutionary information, they primarily depend on evolutionary change patterns without explicitly considering the structural information derived from the dependencies among software classes. Consequently, they do not integrate evolutionary information with structural information to comprehensively measure the change propagation across software classes.

Structural information has also been used for Shotgun Surgery detection independently. Rao and Reddy proposed the Design Change Propagation Probability (DCPP) method to detect this smell and the Divergent Change smell using structural information.[28] DCPP measures structural dependency strength through the metric (C-degree), using method invocations and variable references, and measures direct and indirect class dependencies using an adjacency matrix. Smelly candidates are detected using predefined rules and heuristics applied to this quantified information. However, DCPP captures limited structural information and does not consider the evolutionary co-change behaviour. Conversely, evolutionary methods capture historical co-change behaviour without explicitly measuring the underlying structural information. Thus, existing approaches for Shotgun Surgery detection largely investigate these dimensions independently, emphasising the need to integrate this information using structural and evolutionary information based on its relative importance for more comprehensive change propagation analysis. Our previous approach also supports the effectiveness of this integrated method in the detection of multiple design smells.[29]

Research Methodology

This paper proposes a quantitative approach for shotgun surgery code smell detection. This methodology is carried out in the following steps, as shown in the block diagram in Fig. 1.

1. Extraction of the structural metrics to find the strength of coupling between each pair of classes.

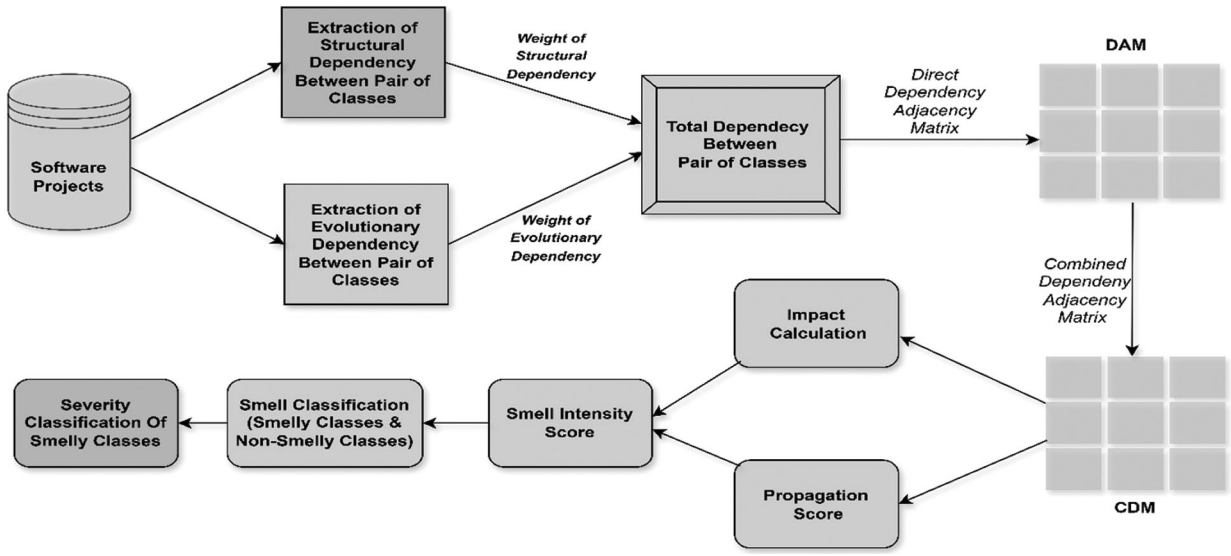


Fig. 1 — Block Diagram for the Proposed Methodology

2. Mining the co-change data of each class pair from each project's version history and extracting the evolutionary information.
3. Integrate structural and evolutionary characteristics by assigning weights to certain features according to their significance. After integrating features based on their relevance, calculate a Total Dependency Score (TDS) between each pair of classes.
4. Construction of a square matrix (size N x N, where N is the number of classes) with structural dependency and evolutionary metrics values between each pair of classes. It is a Direct Dependency Adjacency Matrix (DDAM) used to fill out the TDS score of each pair of classes.
5. Construction of another square matrix of size NxN, where, N is the number of classes denoted as Combined Dependency Adjacency Matrix (CDAM) for the consideration of indirect dependency.
6. Define the heuristic to calculate the impact and propagation score of each class from the CDAM matrix designed above.
7. Calculate the smell intensity score based on the relevance of the impact and propagation score.
8. Classification of the smelly and non-smelly classes based on the thresholds of the smell intensity score.

This methodology follows a sequence of steps for smell detection by integrating the structural and evolutionary dependencies between each pair of classes. First, the source code analysis and version history analysis are performed to calculate the Structural Dependency Score (SDS) for structural measurement and the Change Impact Score (CIS) for evolutionary measurement between each pair of classes, respectively. Each dependency score is then assigned its respective weights and combined to determine the Total Dependency Score (TDS) between each pair of classes. The resulting pairwise dependency scores (TDS) are used to form the Direct Dependency Adjacency Matrix (DDAM), which shows the direct dependencies among classes. Afterwards, indirect dependency measures are included to form the Combined Dependency Adjacency Matrix (CDAM). Using the total dependency value shown in CDAM, the impact score and propagation score are calculated for each class. These two scores are then integrated to determine the smell intensity score, which is used for smell classification in smelly and non-smelly classes. The structural analysis of the source code is limited to the Java source code only, while the evolutionary analysis is language-independent, as it is the class-level information derived from the version control systems. The following section explains the details of methodology.

Structural Dependency

Structural dependencies between each pair of classes are extracted using Javalang,[30] which parses the Java source code into an Abstract Syntax Tree (AST). This AST is analyzed for measuring the six structural dependencies between a class A (source)

and class B (target). This structural information captures the different sources of coupling rather than considering all dependencies as simple method calls:

Inheritance Dependency: When class A extends class B.

Field Dependency: When class A contains the field of class B.

Return Type Dependency: When a method of class A returns class B type.

Parameter Dependency: When a method of class A accepts parameter of class B type.

Method Call Dependency: When method m1 of class A calls the methods m2 of class B.

Static Access Dependency: When static member of class A is accessed by the class B. After assigning weight to each type of dependency the final structural dependency score SDS (A, B) is calculated between Class A and Class B using given formula in Eq. 1.

$$SDS_{(A,B)} = \Sigma\,(Di_{(A,B)} \times w_i) \quad \ldots (1)$$

Here, D_i (A, B) measures the dependency type i between class A and class B, and W_i is its assigned weight.

Evolutionary Dependency

Evolutionary information is extracted from Git history using PyDriller,[31] which uses commit-level file changes, developer information, and date of modification. A pair of classes may change together frequently because modifications in one class may affect the other. To capture this evolutionary behavior, the proposed Change Impact Score (CIS) integrates four factors: co-change count, added and deleted lines (churn), developer overlap, and average time between co-changing commits. These measures are weighted and integrated in Eq. (2). A higher value of CIS shows the stronger co-change association, which compliments the dependencies extracted through static analysis.

$$CIS\,(A,B) = W_1 \times CC\,(A,B) + W_2 \times CH(A,B) + W_3 \times DO(A,B) + W_4 \times \left(\frac{1}{ATCC(A,B)}\right) \quad \ldots (2)$$

Here, in Eq. (2), CC (A, B) counts the commits that change both classes. A and B; Ch (A, B) summarizes the churn (lines added and lines deleted); DO (A,B) counts the developers participating in both classes A and B; and ATCC (A,B) measures the average time gap between commits involving the pair of classes. W_1, W_2, W_3, and W_4 are the weights based on the relative contributions of these factors. This integrated score captures commit frequency, magnitude, developer participation, and timing of evolutionary changes.

Total Dependency Score

SDS and CIS measure the complementary information of change propagation. SDS quantifies the structural dependencies that allow changes to affect other classes; CIS measures their observed co-change behavior during software evolution. Using only one type of information may overlook some relevant relationships. Therefore, the proposed Total Dependency Score (TDS) integrates normalized SDS and CIS scores using appropriate weights, as shown in Eq. (3). Let α and β be the weights assigned to SDS and CIS, respectively, such that $\alpha + \beta = 1$. The assigned weights are based on the relative importance of structural and evolutionary information.

$$TDS_{(A,B)} = \alpha \times SDS_{(A,B)} + \beta \times CIS_{(A,B)} \quad \ldots (3)$$

Here in Eq. (3), normalized SDS (A, B) and CIS (A, B) measure the structural and evolutionary dependencies, respectively; the larger TDS value shows stronger combined dependency.

Construction of DDAM and CDAM Matrix

For a software system having N classes, the Direct Dependency Adjacency Matrix (DDAM) is an N × N matrix where DDAM (A, B) stores the direct pairwise value TDS score; entries with no direct relationships are zero. Measuring direct dependencies alone is not sufficient because a change originated from class A may reach class B through intermediate classes such as class C or class D. Consequently, the Combined Dependency Adjacency Matrix (CDAM) additionally measures the indirect relationship. When a direct path exists, its TDS value is retained. Otherwise, for an indirect path, CDAM (A, B) is calculated using Eq. (4); if there is no path (direct or indirect), the CDAM (A, B) = 0. This measures the analysis of change propagation at the class level beyond immediate neighbours.

$$CDAM_{(A,B)} = 1 - ((1 - DDAM_{(A,C)}) * (1 - DDAM_{(C,B)})) \quad \ldots (4)$$

Here, CDAM (A, B) is the combined dependency score between class A and class B; DDAM (A, C) and

DDAM (C, B) measures the dependencies of class A to class B through intermediate class C.

Heuristic for Smell Detection

The heuristic for smell detection calculates the smell intensity score, based on the two factors, propagation score and impact score, explained below:

Propagation Score

It quantifies how many classes are affected by a class when it undergoes a change. It is calculated by taking the sum of non-zero elements corresponding to the row. The propagation score of class A can be calculated using the formula below in Eq. 5.

$$Propagation\ Score\ (A) = Count\ of\ nonzero\ values\ in\ row\ A \quad \ldots (5)$$

Impact Score

This quantifies the total impact a class has on other classes. It can be calculated by summing the values of each row in the matrix. The impact score for class A can be calculated using the formula in Eq. 6.

$$\text{Impact Score (A)} = \sum(\text{B})\ \text{CDAM}_{[\text{A}]\ [\text{B}]} \quad \ldots (6)$$

Smell Intensity Score

The smell intensity score for each class is a weighted sum of the propagation score and impact score. The smell intensity score for class A can be calculated using the formula below in Eq. 7.

$$Smell\ Intensity\ Score\ (A) = Propagation\ Score\ (A) + Impact\ Score\ (A) \quad \ldots (7)$$

Smell Classification

The smell intensity score of each class defines the impact and propagation intensity of that particular class in the entire project. Based on that, a dynamic threshold has been derived using statistical measures to classify smelly and non-smelly classes.

Results and Discussion

In this section the empirical findings of this work have been explained with the discussion of some research questions relevant to the proposed technique and discussed them one by one.

RQ1: How does the proposed technique detect the Shotgun Surgery smell using structural and evolutionary features?

To answer the RQ1, a sample Java project having six classes (C1–C6) and one month of change history has been taken as a working example. The class interactions can be seen from code snippet in Fig. 2, and SDS is calculated using six structural dependency weights: inheritance 0.5, field 0.2, return type 0.1, parameter 0.1, method call 0.05, and static access 0.05 (as shown in Table 1). The structural direct dependency graph is shown in Fig. 3. The commit history of the working example is presented in Table 2 from which evolutionary information is quantified. This working example explicitly shows the intermediate calculations of the proposed approach.

```
public class ClassC1 extends ClassC2 {
   private ClassC3 fieldC3;

   public ClassC3 getFieldC3() {
       return fieldC3;
    }

   public void setFieldC3(ClassC3 fieldC3) {
this.fieldC3 = fieldC3;
   }

   public void methodC1(ClassC4 paramC4) {
           paramC4.methodC4();
    }
}

public class ClassC2 {
   public static String staticField =    "Static_Access";
   public void methodC2() {
System.out.println("MethodC2 in ClassC2");
   }
}
public class ClassC3 {
           public void methodC3() {
          System.out.println("MethodC3 in ClassC3");
           }
}
public class ClassC4 {
           public void methodC4() {
          System.out.println("MethodC4 in ClassC4");
           }
}
public class ClassC5 {
   public void methodC5() {
          System.out.println("Accessing static field of
          ClassC2");
          System.out.println(ClassC2.staticField);
     }
}
public class ClassC6 {
    public void methodC6(ClassC3 paramC3) {
               paramC3.methodC3();
  }
}
```

Fig. 2 — Code snippet for the class interaction

Table 1 — Class dependencies and their calculated SDS

Class pair	Dependency type	Count	Weight	SDS calculation	Total SDS
C1->C2	Inheritance	1	0.5	1×0.5 =0.5	0.5
C1->C3	Field	1	0.2	1×0.2 =0.2	0.2+0.1+0.1= 0.4
	Return type	1	0.1	1×0.1 =0.1	
	Parameter	1	0.1	1×0.1 =0.1	
C1->C4	Method call	1	0.05	1×0.05 =0.05	0.05+0.1=0.15
	Parameter	1	0.1	1×0.1 =0.1	
C5->C2	Static access	1	0.05	1×0.05 =0.05	0.05
C6->C3	Parameter	1	0.1	1×0.1 =0.1	0.05+0.1= 0.15
	Method call	1	0.05	1×0.05 =0.05	

Table 2 — Commit History of the working example project

Commit	Files modified	Lines added/deleted	Developer	Commit date
C1	File C1.java, File C2.java	10 added, 5 deleted	D1	2023-01-01
C2	File C1.java, File C3.java	20 added, 10 deleted	D2	2023-01-02
C3	File C1.java, File C2.java	15 added, 8 deleted	D1	2023-01-05
C4	File C2.java, File C4.java	25 added, 15 deleted	D3	2023-01-07
C5	File C5.java, File C6.java	30 added, 20 deleted	D1	2023-01-10
C6	File C1.java, File C2.java	5 added, 2 deleted	D2	2023-01-12

Table 3 — Evolutionary metrics calculation for mined commit history

Commit	Files modified	Co-change count	Lines added / deleted	Dev-overlap	Time gap
Commit C1:	C1, C2	1	10/5	D1	—
Commit C2:	C1, C3	1	20/10	D2	—
Commit C3:	C1, C2	2	15/8	D1	4 days
Commit C4:	C2, C4	1	25/15	D3	—
Commit C5:	C5, C6	1	30/20	D1	—
Commit C6:	C1, C2	3	5/2	D1, D2	7 days

The corresponding SDS dependency graph is presented in Fig. 3, and Table 2 shows the commit information for subsequent evolutionary analysis. The integrated CIS score, which uses co-change count, churn measure, developer overlap information, and time gap along with their assumed relative weights and intermediate calculations, is presented in Tables 3 and 4, respectively. The direct dependency graph after calculating the CIS calculation of class pairs, with the corresponding CIS score has been shown in Fig. 4.

SDS and CIS scores are combined to form TDS, using structural and evolutionary weights of 0.4 and 0.6, respectively. The higher value of evolutionary weight shows the maintenance-oriented nature of the Shotgun Surgery smell. The TDS score is normalized and stored as direct relationships in DDAM (Table 5). CDAM then calculated using indirect relationships (Table 6). Class-level propagation score and impact score are calculated using Eqs (5 & 6), while Eq. (7) combines this information into the smell intensity score; Table 7 shows the intermediate scores, normalization, and resultant smelly and non-smelly classes.

For this empirical evaluation the four open-source Apache Java projects—Ant, Camel, Log4j and POI are selected to perform this analysis. The selection criteria of these projects are based on the availability of the source code, change history for evolutionary measurement, and long maintenance history for longitudinal analysis. The brief of the selected software projects is given in Table 8.

The illustrated working example classifies C1 and C2 as smelly classes and C3–C6 as non-smelly classes (Table 7), this research question demonstrates how structural and evolutionary information jointly find the smelly and non-smelly classes.

RQ2: How does the proposed technique perform compared to existing techniques in detecting code smell instances using both structural and evolutionary features?

The methodological differences between the proposed smell detection approach and the two

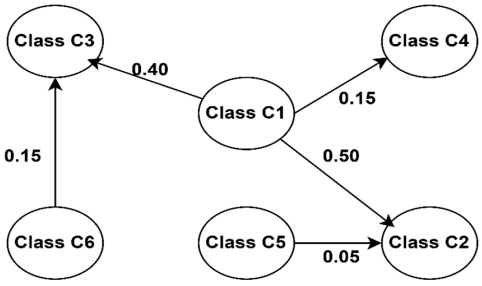


Fig. 3 — Direct dependency graph for SDS

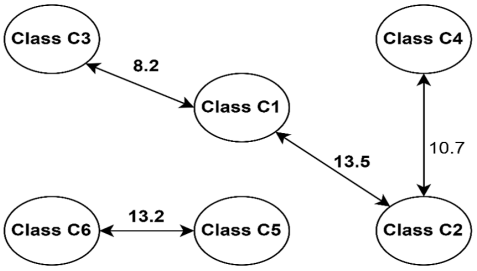


Fig. 4 — Direct dependency graph for CIS

Table 4 — CIS calculation of class pair

File pair	CIS calculation
C1, C2	(0.65×3) + (0.25×45) + (0.05×2) + (0.05/5.5) ≈ 13.5
C1, C3	(0.65×1) + (0.25×30) + (0.05×1) + (0.05/∞) ≈ 8.2
C2, C4	(0.65×1) + (0.25×40) + (0.05×1) + (0.05/∞) ≈ 10.7
C5, C6	(0.65×1) + (0.25×50) + (0.05×1) + (0.05/∞) ≈ 13.2

existing approaches, DCPP and HIST, are listed in Table 9. The DCPP approach is based on the static dependencies measured using C-degree and HIST, which is dependent on the software change history, and the proposed smell detection approach combines SDS and CIS into TDS. The comparison does not only focus on the number of smelly candidates but also on the type of dependency in smell detection.

Across the four analyzed software systems, DCPP approach identified 73 smelly classes and SDS approach identified 64 (12.33% fewer) as shown in Table 10. This difference in the detection of the smelly instances should not be interpreted from detection performance point of view as DCPP uses only method calls and member references, whereas SDS captures a broader range of structural dependencies and using different detection rules.

The existing HIST approach identified 6 smelly instances, while the proposed CIS approach identified 9 (both based on evolutionary information), resulting in a 50% increase in detected smelly instances across the project, as shown in Table 10. The difference in detection is due to coverage of change history information, as HIST only measures the change analysis, whereas CIS incorporates co-change frequency, churn, developer overlap information, and time gap

Table 5 — DDAM for total dependency score

Class	C1	C2	C3	C4	C5	C6
C1	1	2.12	0.78	1.16	0	0
C2	1	1	0.78	0.69	0	0
C3	0	0	1	0	0	0
C4	0.47	0.47	0	1	0	0
C5	0	0	0.21	0	1	0.94
C6	0	0	0.22	0	0.94	1

Table 6 — CDAM for total dependency score

Class	C1	C2	C3	C4	C5	C6
C1	1	2	0.78	0.22	0	0
C2	1	1	0	0.47	0	0
C3	0	0	1	0	0	0
C4	0	0.47	0	1	0	0
C5	0	0	0	0	1	0.94
C6	0	0	0.22	0	0.94	1

Table 7 — Smell classification using smell intensity score of the working example classes

Class	Propagation score	Impact score	Smell intensity score	Normalized score	Smell classification
C1	4	5.06	9.06	1	Smelly
C2	4	3.47	7.47	0.83	Smelly
C3	1	1	2	0.22	Non-Smelly
C4	3	1.94	4.94	0.55	Non-Smelly
C5	3	2.15	5.15	0.57	Non-Smelly
C6	3	2.16	5.16	0.57	Non-Smelly

Table 8 — Details of the software project used for the analysis

Projects	Version	Domain
Apache Ant	1.6	Java library and command line tool
Apache Camel	1.6	An open-source framework that integrates systems by moving data between them
Apache Log4j	1.0	A versatile, feature-rich, efficient logging API and backend for Java
Apache Poi	2.5	A java library for reading and writing microsoft office binary and OOXML file formats

Table 9 — Methodological comparison of the proposed heuristic and the existing techniques for shotgun surgery detection

Parameters	Proposed heuristic	DCPP	HIST
Features	Structural + Evolutionary	Only structural	Only evolutionary
Dependency score	Used a weighted combination of structural and evolutionary features for total dependency	Used C-degree which only exploits the structural dependency	Analyse changes across versions using version history
Dependency level	Consider direct and indirect dependency both	Consider direct and indirect dependency both	Only analyze the historical changes
Propagation	Calculate the propagation score to find the impact on other classes	Does not calculate the propagation of class explicitly	Only counts the frequency of changes
Smell Intensity Score	Weighted integration of structural and evolutionary features to calculate intensity score	C-degree defines shotgun surgery detection	Historical features define the likelihood of smelly candidates
Limitation	Need of both structural and evolutionary information	Missed smells by evolutionary features	Highly dependent on the quality of version history
Suitability	Highly effective for system-required structural and change analysis	Effective for those systems where structural information is sufficient	It is best for systems with extensive version history

Table 10 — Comparison of the existing approach and proposed approach for smell detection

Projects	Structural		Evolutionary		Integrated
	Existing	Proposed	Existing	Proposed	proposed
	DCPP	*SDS*	*HIST*	*CIS*	*TDS*
Apache Ant	44	38	01	01	08
Apache Camel	15	16	02	04	02
Apache Log4j	05	03	01	02	03
Apache Poi	09	07	02	02	05

Table 11 — Overlapping analysis of smelly instances detected by existing and proposed technique

Software Projects	Structural			Evolutionary		
	Smells (D ∩ S)	Smells (D \ S)	Smells (S \ D)	Smells (H ∩ C)	Smells (H\ C)	Smells (C \ H)
Apache Ant	17.14 %	45.71 %	37.14 %	100%	0%	0%
Apache Camel	29.16 %	33.33 %	37.5 %	20 %	20 %	60 %
Apache Log4j	33.33 %	50 %	16.67 %	50 %	0 %	50 %
Apache Poi	33.33 %	41.66 %	25 %	33.33 %	33.33 %	33.33%

between two commits. The proposed approach (CIS) represents broader feature coverage. The integrated approach TDS identified 8, 2, 3, and 5 smelly classes in Ant, Camel, Log4j, and POI, respectively (18 smelly instances overall). Of these, 6 smelly classes were associated with buggy classes (33.33%). This integrated approach is intended to identify the smelly classes with evidence of both structural dependencies and evolutionary co-change information, rather than to maximize the number of flagged smelly candidates. The contrasting DCPP/ SDS and HIST/CIS candidate sets show the complementary behavior of the information used by TDS.

RQ3: What are the common and missed instances detected by both the proposed and existing approaches?

To answer the RQ3, while performing smell detection, we computed the following overlapping metrics defined in (Eqs 8 & 9). Let smells (Pi) and smells (Ei) represent the sets of smelly instances detected by the proposed and existing techniques, respectively. The intersection (Pi ∩ Ei) denotes common

Table 12 — Distribution of smelly and buggy instances across software projects

Projects	Technique	S & B	S & NB	NS & B	NS & NB
Apache Ant	DCPP	14	30	78	229
	SDS	10	28	82	231
	CIS	01	00	91	259
	TDS	02	06	90	253
Apache Camel	DCPP	02	13	186	764
	SDS	02	14	186	763
	CIS	01	03	187	774
	TDS	01	01	187	776
Apache Log4j	DCPP	01	04	33	97
	SDS	01	02	33	99
	CIS	01	01	33	100
	TDS	01	02	33	99
Apache Poi	DCPP	03	06	245	379
	SDS	02	03	246	382
	CIS	00	02	248	383
	TDS	02	03	246	382

smelly instances, whereas Pi \ Ei and Ei \ Pi denotes smelly instances detected by one approach but missed by the other.

$$\text{Smells (Pi} \cap \text{Ei)} = \frac{\text{Detect (Pi} \cap \text{Ei)}}{\text{Detect (Pi} \cup \text{Ei)}} * 100 \quad \ldots (8)$$

$$\text{Smells (Pi} \setminus \text{Ei)} = \frac{\text{Detect (Pi)} \setminus \text{Detect (Ei)}}{\text{Detect (Pi)} \cup \text{Detect (Ei)}} * 100 \quad \ldots (9)$$

The findings of smelly candidates are listed in Table 11 using overlapping metrics and unique candidates for structural (DCPP versus SDS) and evolutionary (HIST versus CIS) comparisons. The average of DCPP–SDS overlap across selected projects was 28.24%, ranging from 17.14% in Apache Ant to 33.33% in Apache Log4j and Apache POI. The findings using evolutionary information also varied: the HIST and CIS approaches reported 100% common smelly candidates in Ant, compared with 20% in Camel, 50% in Log4j, and 33.33% in POI. The distinct candidate sets reflect the differences in the measured dependencies and heuristics used by the approaches; however, overlap metrics alone do not establish the accuracy of the detectors.

RQ4: How many of the detected smelly instances lead to buggy instances, and what is their distribution with their buggy nature?

To answer this research question, we perform bug association analysis of the detected smelly instances using the benchmark bug prediction dataset. The mapping of smelly instances found by the proposed approach, the notation of S & B (Smelly and Buggy), S & NB (Smelly and Not Buggy), NS & B (Non-smelly and Buggy), and NS & NB (Not Smelly and Not Buggy) used are given in Table 12. This mapping shows that the DCPP method found the most smelly and buggy instances 14 and Smelly and not buggy instances 30 in Ant-1.6. In comparison, the SDS method only found 10 Smelly and buggy instances and 28 smelly and non-buggy instances. The DCPP method also found 78 non-smelly but buggy instances, while the SDS method detected 82 such instances.

The experimentation performed to answer this research question reveals that all smelly instances are not buggy, but they are they violate the object-oriented design principles from the experimentation on the above four projects using integration of structural and evolutionary features 25%, 50%, 33.33%, and 40% instances for Apache Ant, Apache Camel, Apache Log4j, and Apache Poi respectively are leads to buggy out of the detected smelly instances.

Conclusions

This paper proposed an integrated method for Shotgun Surgery code smell detection using structural and evolutionary information of Java-based object-oriented software systems. The bug association examination at the project level reveals that 25%, 50%, 33.33%, and 40% of the smelly instances in Apache Ant, Apache Camel, Apache Log4j, and Apache POI, respectively, were associated with bugs. These findings show the maintenance can be more effective and efficient by considering the combination of structural and evolutionary information. However, as this empirical evaluation covers four maintained Apache Java projects, and these findings may not

generalize to smaller systems and other languages or procedural paradigms. The proposed heuristic requires sufficiently complete source code and a long change history. The manually assigned dependency weights may be context-sensitive, and selected repository quality and weight selection may therefore affect the classification of smelly and non-smelly instances. In future some limitations of this work can be addressed using objective weight derivation through optimization technique, statistical learning or supervised models. Sensitivity analysis may also be performed using different weight combinations to determine the stability of the detected smelly instances. The empirical evaluation can be extended to industrial object-oriented software systems in other languages.

References

1 Fowler M, *Refactoring: Improving the Design of Existing Code*, **2nd edn** (Addison-Wesley Professional), 2018

2 Tahir A, Dietrich J, Counsell S, Licorish S A & Yamashita A, A large scale study on how developers discuss code smells and anti-pattern in Stack Exchange sites, *Inf Softw Technol*, **125** (2020) 106333, doi: 10.1016/j.infsof.2020.106333.

3 Zazworka N, Vetro' A, Izurieta C, Wong S, Cai Y & Seaman C, *et al*., Comparing four approaches for technical debt identification, *Softw Qual J*, **22** (2014) 403–426, doi: https://doi.org/10.1007/s11219-013-9200-8.

4 Arcelli Fontana F, Dietrich J, Walter B, Yamashita A & Zanoni M, Antipattern and code smell false positives: Preliminary conceptualization and classification, *Proc 23rd IEEE Int Conf Softw Anal Evol Reeng (SANER)*, (Suita, Osaka, Japan), (2016) 609–613, doi: 10.1109/SANER.2016.84.

5 Mäntylä M V, Vanhanen J & Lassenius C, Bad smells – humans as code critics, *Proc 20th IEEE Int Conf Softw Maint (ICSM)*, (Chicago, IL, USA), (2004) 399–408, doi: 10.1109/ICSM.2004.1357825.

6 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)** (2015) 462–489, doi: 10.1109/TSE.2014.2372760.

7 Prokić S, Luburić N, Slivka J & Kovačević A, Prescriptive procedure for manual code smell annotation, *Sci Comput Program*, **238** (2024) 103168, doi: 10.1016/j.scico.2024.103168.

8 Slivka J, Luburić N, Prokić S, Grujić K-G, Kovačević A, Sladić G & Vidaković D, Towards a systematic approach to manual annotation of code smells, *Sci Comput Program*, **230** (2023) 102999, doi: 10.1016/j.scico.2023.102999.

9 Chidamber S R & Kemerer C F, A metrics suite for object oriented design, *IEEE Trans Softw Eng*, **20(6)** (1994) 476–493, doi: 10.1109/32.295895.

10 McCabe T J, A complexity measure, *IEEE Trans Softw Eng*, **SE-2(4)** (1976) 308–320, doi: 10.1109/TSE.1976.233837.

11 Singh M & Chhabra J K, Improved software fault prediction model based on optimal features set and threshold values using metaheuristic approach, *SN Comput Sci*, **4(6)** (2023) 770, doi: 10.1007/s42979-023-02217-x.

12 Lanza M & Marinescu R, *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems* (Springer, Berlin, Heidelberg), 2006, doi: 10.1007/3-540-39538-5.

13 Murphy-Hill E & Black A P, An interactive ambient visualization for code smells, *Proc 5th Int Symp Softw Vis (SOFTVIS)*, ACM, (Salt Lake City, UT, USA), (2010) 5–14, doi: 10.1145/1879211.1879216.

14 dos Reis J P, Brito e Abreu F & de Figueiredo Carneiro G, Code smells detection 2.0: crowdsmelling and visualization, *Proc 12th Iberian Conf Inf Syst Technol (CISTI)*, IEEE, (Lisbon, Portugal), (2017) 1–4, doi: 10.23919/CISTI.2017.7975961.

15 PMD: source code analyzer, https://pmd.github.io/

16 Checkstyle: Java code quality tool, https://checkstyle.sourceforge.io/ J

17 Deodorant: Code smell detection and refactoring tool, https://github.com/tsantalis/JDeodorant/

18 Arcelli Fontana F & Zanoni M, Code smell severity classification using machine learning techniques, *Knowl Based Syst*, **128** (2017) 43–58, doi: 10.1016/j.knosys.2017.04.014

19 Arcelli Fontana F, Mäntylä M V, Zanoni M & Marino A, Comparing and experimenting machine learning techniques for code smell detection, *Empir Softw Eng*, **21(3)** (2016) 1143–1191, doi: 10.1007/s10664-015-9378-4.

20 Tsantalis N, Chaikalis T & Chatzigeorgiou A, JDeodorant: identification and removal of type-checking bad smells, *Proc 12th Eur Conf Softw Maint Reeng (CSMR)*, IEEE Computer Society, (Athens, Greece), (2008) 329–331, doi: 10.1109/CSMR.2008.4493342.

21 Dewangan S, Rao R S, Mishra A & Gupta M, Code smell detection using ensemble machine learning algorithms, *Appl Sci*, **12(20)** (2022) 10321, doi: 10.3390/app122010321.

22 Liu H, Jin J, Xu Z, Zou Y, Bu Y & Zhang L, Deep learning based code smell detection, *IEEE Trans Softw Eng*, **47(9)** (2021) 1811–1837, doi: 10.1109/TSE.2019.2936376.

23 Zhang Y, Ge C, Hong S, Tian R, Dong C & Liu J, DeleSmell: code smell detection based on deep learning and latent semantic analysis, *Knowl Based Syst*, **255** (2022) 109737, doi: 10.1016/j.knosys.2022.109737.

24 Khomh F, Di Penta M & Guéhéneuc Y-G, An exploratory study of the impact of code smells on software change-proneness, *Proc 16th Working Conf Reverse Eng (WCRE)*, IEEE, (Lille, France), (2009) 75–84, doi: 10.1109/WCRE.2009.28.

25 Hozano M, Garcia A, Fonseca B & Costa E, Are you smelling it? Investigating how similar developers detect code smells, *Inf Softw Technol*, **93** (2018) 130–146, doi: 10.1016/j.infsof.2017.09.002.

26 Saranya G, Nehemiah H K & Kannan A, Hybrid particle swarm optimisation with mutation for code smell detection, *Int J Bio Inspired Comput*, **12(3)** (2018) 186–195, doi: 10.1504/IJBIC.2018.094624.

27 Fu S & Shen B, Code bad smell detection through evolutionary data mining, *Proc 9th ACM/IEEE Int Symp Empir Softw Eng Meas (ESEM)*, IEEE Computer Society, (Beijing, China), (2015) 41–49, doi: 10.1109/ ESEM. 2015.7321194.

28 Rao A A & Reddy K N, Detecting bad smells in object oriented design using design change propagation probability matrix, *Proc Int Multiconf Eng Comput Sci (IMECS),* Vol I, (Hong Kong), (2008) 1001–1007.

29 Sharma K & Chhabra J K, Software design smells prediction using machine learning with evolutionary and structural metrics of source code, *J Comput Lang*, **86** (2026) 101373, doi: 10.1016/j.cola.2025.101373.

30 javalang: pure Python Java parser and tools, https:// github.com/c2nes/javalang

31 PyDriller: Python framework to analyse Git repositories, https://github.com/ishepard/pydriller