Discover Computing

RESEARCH Open Access

Ontology driven cross language design smell detection through semantic reasoning

Sabiyyah Sabir[1,2], Ghulam Rasool[1], Tariq Umer[1], Harun Elkiran[3*] and Jawad Rasheed[4*]

*Correspondence:
Harun Elkiran
harun.elkiran@medipol.edu.tr
Jawad Rasheed
jawad.rasheed@yildiz.edu.tr
[1]Department of Computer Science, COMSATS University Islamabad, Lahore Campus, Lahore 54000, Pakistan
[2]Department of Artificial Intelligence, The University of Lahore, Lahore 54000, Pakistan
[3]Faculty of Engineering and Natural Sciences, Istanbul Medipol University, Istanbul 34810, Türkiye
[4]Clean Energy Technologies Institute, Yıldız Technical University, Istanbul 34220, Türkiye

Abstract

Design smells are structural flaws that, if not detected early, can impair software quality and increase maintenance costs. Traditional detection approaches are language-specific and rely on low-level metrics, making them ineffective for heterogeneous, multilingual systems. This paper presents Ontology-Driven Design Detector (OD3), a language-agnostic framework for detecting design smells at the design stage. OD3 reverse-engineers source code from Java, C++, C#, Ruby, and Python into a unified semantic ontology and applies rule-based reasoning to identify violations of Object-Oriented (OO) design principles. Its key contributions are: (i) detecting design smells in multilingual software projects, (ii) defining semantic representations of smells to improve interpretability, (iii) reducing incorrect detections through context-aware reasoning, (iv) introducing a catalog of cross-language design smells, and (v) validating the approach on 10 multilingual systems, achieving a precision and recall of 0.91 and 0.92, respectively. These results support scalable, early-stage detection of design flaws across heterogeneous software systems while maintaining high semantic accuracy. The proposed framework is designed to operate efficiently within High-Performance Computing (HPC) environments, where parallel ontology reasoning and distributed semantic processing facilitate the analysis of large multilingual software repositories. Consequently, the framework contributes to Sustainable Development Goal (SDG) 9 by supporting intelligent, scalable software engineering infrastructure.

Keywords Ontology engineering, Cross-language software engineering, Codebases, Semantic reasoning, Software quality assurance, Distributed semantic computing, Large-scale software analytics

1 Introduction

Software design smells are subtle structural issues that can degrade software quality and complicate development. Traditional detection approaches rely on language-specific source code analysis, which limits scalability and hinders reuse across programming languages [1, 2, 3]. Because these techniques depend on language-bound syntactic and semantic features, models trained for one language (e.g., Java) often fail to generalize to others like C# or Python without costly re-engineering [4, 5, 6, 7], leading to reduced portability and higher maintenance overheads [8, 9]. A practical way to overcome this

© The Author(s) 2026. **Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

dependency is to detect smells at the design stage using language-independent models of Unified Modeling Language (UML) [10]. Early detection of issues such as God Class or Shotgun Surgery helps reduce technical debt before implementation [11, 12]. However, UML's limited formal semantics restrict automated reasoning and reduce detection accuracy [13, 14]. Ontology-based models address these limitations by offering formal, machine-interpretable representations of design elements and constraints [15, 16]. Through semantic rules and inference, they can detect deeper design-principle violations and avoid the incorrect detections common in metric-based approaches [14, 17, 18, 19, 20]. Research shows that ontology-driven detection improves accuracy and supports continuous reasoning throughout the Software Development Lifecycle(SDLC) [7, 10, 11, 21].

As software repositories continue to grow in size, complexity, and linguistic diversity, the computational demands associated with semantic knowledge extraction, ontology population, ontology matching, and Description Logic reasoning increase considerably. An HPC environment provides an effective platform for addressing these computational challenges by enabling distributed processing, parallel semantic reasoning, and scalable knowledge management across large software repositories. Although the proposed framework does not introduce a new HPC algorithm, its ontology-driven architecture is inherently well-suited for HPC-enabled deployment, in which multiple software projects and ontologies can be processed concurrently, thereby significantly improving scalability, throughput, and practical applicability in industrial software engineering environments.

Beyond computational scalability, the proposed ontology-driven framework aligns with Sustainable Development Goal (SDG)-9 (Industry, Innovation and Infrastructure) by supporting sustainable software engineering practices through early design quality assessment and reusable semantic knowledge models. Detecting design deficiencies during the early stages of software development reduces future maintenance effort, minimizes technical debt, and improves software reliability. Furthermore, the combination of ontology-driven reasoning and HPC-enabled execution supports scalable software quality assurance for heterogeneous software repositories, thereby contributing to resilient digital infrastructure and intelligent software engineering processes.

Software smells can occur at multiple abstraction levels, including architectural, design, code, and requirement smells, as well as less-discussed types such as test or comment smells. These terms are often used interchangeably in the literature, leading to the misclassification of issues like Blob or Spaghetti Code as design smells, even though they do not align with design-level concerns. Architectural smells represent high-level structural flaws in system architecture, often caused by poor integration of frameworks or components. Smells such as Dense Structure and God Component make systems rigid and difficult to maintain [22, 23]. Design smells arise from violations of OO principles at the class level. They indicate weak abstraction, poor encapsulation, or misplaced responsibilities. Examples include Broken Modularization and Multifaceted Abstraction, which signal unclear modular boundaries or overloaded classes [24]. Code smells are low-level implementation issues that affect readability and maintainability, such as Long Method or long parameter lists. Addressing these smells reduces technical debt and improves overall code clarity [25]. Requirement smells emerge from inconsistent requirements and can cause downstream design and implementation issues. Ambiguous Requirement is a common example that has received increased attention in recent studies [26].

Table 1 summarizes the key differences between design smells and code smells to clarify the conceptual distinction.

The evolution of formal ontologies, such as the Web Ontology Language (OWL) and the Resource Description Framework (RDF), enables machine-interpretable design rules that can capture nuanced architectural knowledge [12, 27]. When integrated into early-stage development workflows, these ontologies provide intelligent support systems that can identify and reason about complex design flaws in a language-neutral manner. This positions ontology-based approaches as a critical advancement in the pursuit of sustainable, language-independent software design quality assurance [28, 29]. Keeping in view the above context, we address the following research questions:

RQ1 What limitations arise from using language-specific code metrics in design smell detection, and how do these limitations affect the applicability of such techniques across software systems developed in heterogeneous programming languages?
Existing detection approaches are tightly coupled with language-specific datasets and tools because they use code-level metrics, making them dependent on the syntax, structure, and semantics of a particular programming language. For instance, a detection technique developed to identify smells in C# projects cannot be applied to Java codebases unless it is adapted to understand Java's constructs and patterns. This results in limited generalizability, increased maintenance effort to support multiple languages, and a lack of unified frameworks for cross-language smell detection.

RQ2 Does detecting design smells at the design stage allow for a programming language-independent detection approach applicable to diverse software systems?
Yes, detecting design smells at the design stage allows for a programming language-independent approach because it focuses on abstract design artifacts, such as UML diagrams, rather than language-specific source code. Since UML models represent system structure and behaviour at a higher level of abstraction, they are not tied to any particular programming language, making the detection approach applicable across diverse software systems regardless of their implementation language.

RQ3 Can ontology-based representations outperform UML in detecting design smells at the design stage by leveraging formal semantics and reasoning capabilities?
Yes, ontologies can perform better smell detection at the design stage compared to UML alone because they provide formal semantics, reasoning capabilities, and rule-based inference that UML lacks. While UML visually models system structure, it does not

Table 1 Difference between design smells and code smells

Aspect	Design Smells	Code Smells
Granularity	Design/Class Level	Implementation/Code Level
Focus	OO principles and architecture quality	Source code quality and readability
Causes	Violation of Abstraction, Modularity, Encapsulation, Hierarchy	Poor coding practices and improper usage of programming constructs
Impact	Affects maintainability, extensibility, and reusability	Affects readability, maintainability, and complexity
Detection Stage	Early design phase or class structure analysis	Implementation and maintenance phase
Refactoring Scope	Structural/design-level refactoring	Code-level refactoring
Examples	Multifaceted Abstraction, Broken Modularization	Long Method, Long Parameter List

inherently support consistency checking or semantic interpretation of design principles. Ontologies, on the other hand, allow encoding of design smells as formal axioms and enable automated reasoning to detect semantic violations, such as responsibility overload or poor abstraction, that are not evident through static UML diagrams. This semantic enrichment leads to more precise, consistent, and context-aware detection of design smells during early stages of the SDLC.

RQ4 Can ontology-based detection reduce false positives in identifying design smells like God Class by incorporating the semantic intent of a class's behavior?
Traditional detection techniques often rely on hardcoded metrics, such as line count or number of responsibilities, which can lead to spurious detections. For example, a class with many print statements may exceed the metric thresholds but still adhere to the single-responsibility principle. Ontology-based approaches overcome this limitation by modeling the semantic behavior of classes, allowing for reasoning about their actual intent. This enables more accurate and context-aware detection of design smells, such as correctly identifying whether a class is truly a God Class. The key contributions of this work are as follows:

- Identified limitations of language-specific, metric-based smell detection and their poor generalizability across heterogeneous systems (RQ1).
- Proposed a language-agnostic, design-stage framework independent of source code syntax (RQ2).
- Compared UML-based and ontology-based approaches, highlighting the semantic advantages of ontologies (RQ3).
- Developed an ontology-driven model with rule-based reasoning for consistent smell detection (RQ3).
- Reduced misclassifications by capturing design intent and behavior (RQ4).
- Defined a cross-language semantic smell catalog for multilingual systems (RQ1, RQ2).
- The first HPC-enabled ontology-driven cross-language design smell detection framework supporting distributed semantic reasoning and scalable analysis of multilingual software repositories.

The rest of this work is structured as follows: Section II covers related work; Section III discusses the research methodology; and Section IV presents results and discussion. The paper concludes with future directions in Section V.

2 Related work

Design smell detection has been a growing area of research within software quality assurance, with a primary focus on code-level analysis using language-specific heuristics. Traditional approaches detected design smells, such as God Class, Feature Envy, and Shotgun Surgery, by using static analysis tools that operate directly on source code. Moreover, smell detection has relied on language-specific static analysis tools such as DECOR and JDeodorant, which identify issues like God Class and Shotgun Surgery by analyzing abstract syntax trees and code metrics. However, these approaches are often limited to single-language ecosystems such as Java or C#, making them less effective in today's increasingly multilingual software environments [30, 31]. To overcome this,

researchers have explored representations such as UML diagrams and ontology language-agnostic [6, 9, 11]. UML-based detection enables early identification of structural flaws through class and sequence diagrams, although limitations like lack of formal semantics and limited automation persist [12, 32].

Ontology-based approaches address this by using semantic reasoning over OWL-based models to detect smells like High Coupling and Low Cohesion, offering more precise, context-aware detection [7, 9, 10, 20]. Recent efforts have explored code-smell detection in heterogeneous, multilingual systems using various techniques. Ansari and Moiz [1–3] proposed static and empirical detectors for multi-language smells, ranging from hybrid web stack interactions (JavaScript–HTML–Java) to JNI misuse (Java–C/C++). They validated new smell types in open-source datasets. Lima et al. [4] applied transfer learning to improve smell detection across Java, Python, and JavaScript. At the same time, Valdivia and Robles [5] analyzed the evolution of smell in polyglot projects, showing its persistence at language boundaries. Buzatu [6] addressed data-level smells in internationalized code. Architectural-level detection across Java, Go, and Python microservices was studied by González and Spinellis [7], and Saini and Gabrani [8] introduced SmellMiner for mining smells from polyglot GitHub repositories. Chowdhury and Roy [9] used graph neural networks on abstracted multilingual code, Daka and Fraser [10] employed language-agnostic AST embeddings, and Tan and Sajnani [11] detected smells via cross-language clone patterns. Finally, Zhou et al. [12] proposed a metric-free, semantic approach supporting C++, Python, and Rust. Table 2 summarizes detection approaches for datasets of a single programming language.

The following points can be concluded based on the information present in the literature:

- Existing techniques focus more on code smells, not design-level issues in single programming language codebases.
- Existing design smell detection tools are tightly coupled to specific programming languages and lack unified, cross-language frameworks for broader applicability.
- Ontology-based methods are not yet integrated into mainstream tools and remain largely untested on large-scale, real-world systems, limiting their practical impact.
- Design-stage smell detection is still underexplored despite its potential for early intervention and language-independent analysis using ontological models.

Table 2 Summary of current research work

Detection Approach	Smells Detected	Dataset Language
Deep learning neural networks [33]	Long Method, God Class	Java
Change history analysis historical metrics [34]	God Class, Feature Envy, Long Method	Java
Machine learning [35]	Data Class, Long Method, Feature Envy, Large Class	Java
Rule-based [36]	Code clone-related smells	Java, C, C++, Python
Code metrics [37]	God Class, Data Class	Java
Rule-based [38]	God Class, Data Class, Feature Envy	Java
Maintainability metrics [39]	God Class, Long Method, Feature Envy	Java
Metric-based statistical analysis [40]	God Class, Shotgun Surgery, Feature Envy	Java
Transformer-based Deep Learning [41]	Long Method, God Class, Feature Envy	Java
Inter-language Rule-based Detection [42]	Inter-language Design Smells	Python, C/C++
Transfer Learning [43]	Code Smells, Bugs, Harmful Code	Java, Python, JavaScript

- Current approaches often overlook the semantic intent and behavioral role of design elements, leading to inaccurate classifications and a higher rate of detection errors.
- Ontology-based models haven't been validated for multilingual datasets.

While numerous studies have investigated ontology-based software analysis, very limited attention has been devoted to executing semantic reasoning pipelines within HPC environments. Existing approaches largely evaluate ontology reasoning on standalone systems, without considering the computational requirements of large-scale multilingual software repositories. This motivates the development of scalable, ontology-driven frameworks that leverage distributed semantic processing for industrial software analytics. Unlike existing ontology-based approaches, which primarily focus on detecting a limited set of design smells within individual programming languages or isolated semantic models, the proposed OD3 introduces a unified semantic reasoning framework that integrates reverse engineering, ontology construction, ontology matching, and rule-based inference into a single language-independent detection pipeline. Rather than relying solely on ontology representation, the framework combines OWL semantic modeling, Semantic Web Rule Language (SWRL) reasoning, and ontology matching to infer instances of design smells from conceptual software knowledge. This semantic reasoning capability enables context-aware interpretation of OO design principles, improves the detection of implicit design violations, and reduces dependence on language-specific metrics. Consequently, OD3 extends existing ontology-driven approaches by providing a scalable, semantically richer framework for multilingual design-smell detection.

In summary, existing research highlights critical gaps in current design smell detection approaches, particularly their reliance on language-specific tools, limited semantic reasoning, and limited applicability at the design stage. The absence of unified, ontology-driven frameworks and standardized benchmarks further hinders progress in this area. Addressing these gaps is essential to enable early, accurate, and language-independent detection of design flaws. This work aims to address these challenges by leveraging ontology-based representations to enhance generalizability, semantic precision, and practical applicability across diverse software systems.

3 Proposed methodology

In this study, we adopt the ontology-enabled detection methodology proposed in [32] and extend its applicability to datasets comprising software systems implemented in multiple programming languages, including Java, C#, Python, and C++. The proposed methodology emphasizes early detection of design smells at the design stage by leveraging ontology-based modeling, design-level metrics, and automated reasoning as follows:

3.1 Dataset collection

To further support our research questions, we intentionally selected datasets implemented in different OO programming languages, ensuring diversity in language constructs, design patterns, and architectural styles, as shown in Table 3. This cross-language selection enables us to evaluate the generalizability and language independence of our ontology-based design smell detection approach. The chosen projects include software systems developed in Java, C#, Python, Ruby, and C++, all of which are actively maintained and widely used in the research community. The selected repositories collectively represent large industrial software systems containing thousands of classes and

Table 3 Selected datasets implemented in different OO programming languages

Sr	Language	Project	KLOC	Classes	First Commit	Last Commit	Open-source /Commercial
1	C++	bitcoin/bitcoin	~450	650+	Jan 2009	Oct 2025	Open-Source
2		Microsoft/ calculator	~250	300+	Mar 2019	Oct 2025	
3	Java	spring-projects/ spring-framework	~800	4,500+	Jan 2005	Oct 2025	
4		elastic/ Elasticsearch	~1,300	5,000+	Feb 2010	Oct 2025	
5	Python	django/django	~900	3,800+	Jul 2005	Oct 2025	
6		pallets/flask	~55	180+	Apr 2010	Oct 2025	
7	C#	dotnet/ aspnetcore	~2,000	6,000+	Dec 2014	Oct 2025	
8		PowerShell/ PowerShell	~1,000	2,500+	Aug 2016	Oct 2025	
9	Ruby	rails/rails	~440	4,000+	Jul 2004	Oct 2025	
10		Github/gemoji	~10	~50+	Feb 2013	Apr 2024	

extensive inheritance relationships, making them representative workloads for HPC-enabled semantic analysis and large-scale ontology reasoning.

3.2 Design smells specification

The design smells used in this study are based on the comprehensive classification introduced by Ganesh et al. [9], which includes 46 distinct smells, 82 identified variants, and 176 aliases. For this research, we selected a subset of those smells as outlined in Table 4. These particular smells were chosen due to their frequent occurrence in real-world software systems and their widespread recognition and use in prior empirical studies, making them well-suited for evaluation and experimentation.

The selected design smells were chosen based on three criteria: (i) their frequent occurrence in empirical software engineering studies, (ii) their direct relationship with fundamental OO quality attributes, including abstraction, modularity, hierarchy, and encapsulation, and (iii) their suitability for formal semantic representation using ontology axioms and SWRL reasoning rules. Collectively, these smells provide comprehensive coverage of the major categories of design degradation observed in OO software systems.

The detection rules presented in Table 4 are expressed using first-order logical predicates and ontology-based semantic notation. The predicate Cls(c) denotes that c is a class, whereas Fld(c, f) and Mth(c, m) represent the attributes and methods associated with a class, respectively. Structural relationships, including Sub, Sup, Anc, Dep, and Gen, denote subclass, superclass, ancestor, dependency, and generalization relationships. Similarity predicates such as Sim and SimStr evaluate semantic or structural similarity between classes, while θ represents predefined threshold values used for smell identification. These predicates are evaluated using ontology axioms and SWRL rules to infer the presence of design-smell instances.

Using the formal detection rules outlined in Table 4, we further interpret the selected design smells and explain the mechanisms used to detect them. For instance, the Deep Hierarchy smell is characterized by scenarios in which a subclass inherits a disproportionately large number of methods relative to its abstract parent. This imbalance often signals poor abstraction or excessive specialization, leading to hierarchies that are

Table 4 Selected subset of smells under study

Design Smell Name	Definition	Detection Axioms
Multifaceted Abstraction	A class handles multiple unrelated responsibilities.	$\exists c(Cls(c)\wedge\exists f, m(Fld(c, f)\wedge Mth(c, m)\wedge(f>7)\wedge(m>7))\wedge\neg LC(c))\Longrightarrow MA(c)$
Unutilized Abstraction	An abstraction is defined but rarely, if ever, used.	$\exists c, s(Cls(c)\wedge\neg Sub(c, s)\wedge\neg Sup(c, s)\wedge(s=0))\Longrightarrow UA(c)$
Unnecessary Abstraction	An abstraction exists without functional necessity.	$\forall c(Cls(c)\wedge\exists fFld(c, f)\wedge\exists m(Mth(c, m)\wedge m=0)\Rightarrow UA(c))$
Large Abstraction	A class is too large or too complex.	$\forall c(Cls(c)\wedge\exists Sc(\lvert Sc\rvert>20)\Rightarrow LA(c))$
Passive Abstraction	A class lacks meaningful attributes or behavior.	$\forall c(Cls(c)\wedge\Phi(c)=\{f, m\}\wedge f=0\wedge m=0\Rightarrow PA(c))$
Duplicate Abstraction	Multiple classes represent the same design concept.	$\forall c(Cls(c)\wedge\exists d(Cls(d)\wedge d\neq c\wedge SimStr(c, d)\geq\theta DA)\Rightarrow DA(c))$
Wide Hierarchy	A class has too many direct subclasses.	$\forall c(Cls(c)\wedge\exists Sc(\lvert Sc\rvert>10)\Rightarrow WH(c))$
Deep Hierarchy	The inheritance chain is excessively long.	$\forall c, c1(Cls(c)\wedge Mth(c, m)\wedge Sub(c1,c)\wedge Mth(c1,2\ m)\Rightarrow DH(c))$
Multipath Hierarchy	A class inherits through multiple hierarchical paths.	$\forall c, p,gp(Cls(c)\wedge Par(c, p)\wedge GPar(c, gp)\Rightarrow MPH(c))$
Cyclic Hierarchy	Classes form circular inheritance relationships.	$\forall c(Cls(c)\wedge\exists d(Cls(d)\wedge Anc(c, d)\wedge Anc(d, c)\wedge c\neq d)\Rightarrow CH(c))$
Missing Hierarchy	Similar classes lack a common abstraction.	$\forall c1\forall c2(Cls(c1)\wedge Cls(c2)\wedge c1\neq c2\wedge Sim(c1,c2)\ \wedge\neg\exists p(Cls(p)\wedge Gen(p, c1)\ \wedge Gen(p, c2))\Rightarrow MG(c1)\wedge MG(c2))$
Deficient Encapsulation	Internal data is unnecessarily exposed.	$\forall c(Cls(c)\wedge\exists f(Fld(c, f)\wedge Vis(f)=public)\Rightarrow DE(c))$
Insufficient Modularization	A class is not properly decomposed.	$\forall c(Cls(c)\wedge Mc=\{M\alpha,M\beta\}\wedge M\gamma(c)=0\wedge M\delta(c)=0\Rightarrow IM(c))$
Broken Modularization	A class has excessive dependencies on both incoming and outgoing sides.	$\forall c(Cls(c)\wedge NDep(c)>\theta IM\Rightarrow IM(c))$
Hub-like Modularization	Responsibilities are fragmented across multiple modules.	$\forall c(Cls(c)\wedge(DegIn(c)+DegOut(c))\geq\theta HM\wedge\exists d1\exists d2(Cls(d1)\wedge Cls(d2)\wedge d1\neq d2\wedge Dep(d1,c)\wedge Dep(c, d2))\Rightarrow HM(c))$

difficult to manage and evolve. To identify this smell, the method hierarchy is analyzed starting from the abstract superclass. Each subclass is examined recursively to detect cases of considerable disparity in method inheritance. Subclasses that accumulate significantly more methods than their base classes are flagged as potential violations of sound design practices. Figure 1 illustrates the proposed HPC-enabled semantic workflow. Although the semantic detection methodology remains independent of the execution platform, the modular architecture naturally supports distributed deployment in HPC environments, enabling ontology extraction, reasoning, and matching to run concurrently across multiple multilingual software repositories.

Figure 1 illustrates the complete semantic detection workflow. Source code is first reverse-engineered into design models, which are then transformed into OWL ontologies. These ontologies are subsequently populated into a semantic knowledge base where ontology matching establishes conceptual correspondences with the reference ontology. Semantic reasoning, supported by OWL axioms and SWRL rules, evaluates OO design constraints to infer design smell instances. The modular architecture enables each stage to operate independently while collectively forming a unified language-independent detection pipeline suitable for large-scale software repositories. Although the proposed semantic workflow is platform-independent, each stage—including ontology extraction, knowledge base construction, ontology matching, and semantic reasoning—can be executed in an HPC environment to improve computational efficiency. Independent

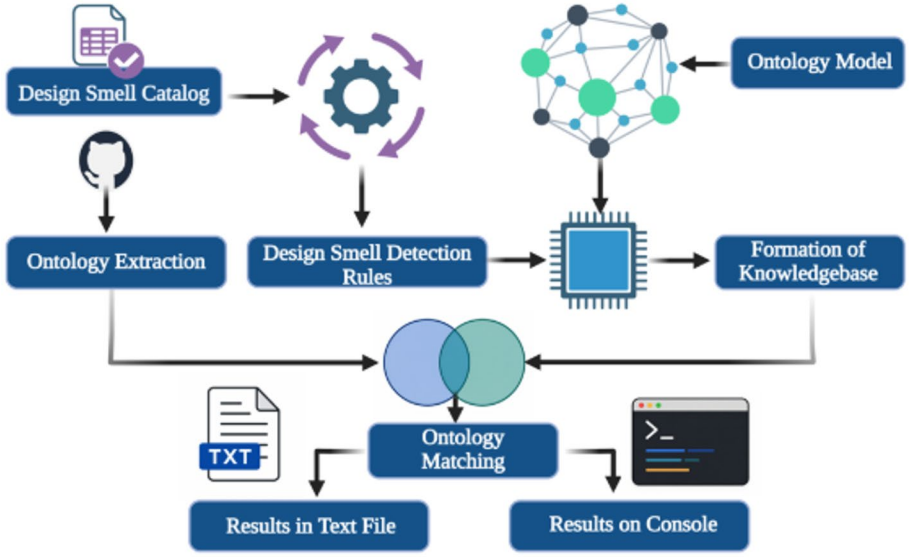


Fig. 1 HPC-enabled semantic workflow for ontology-based cross-language design smell detection

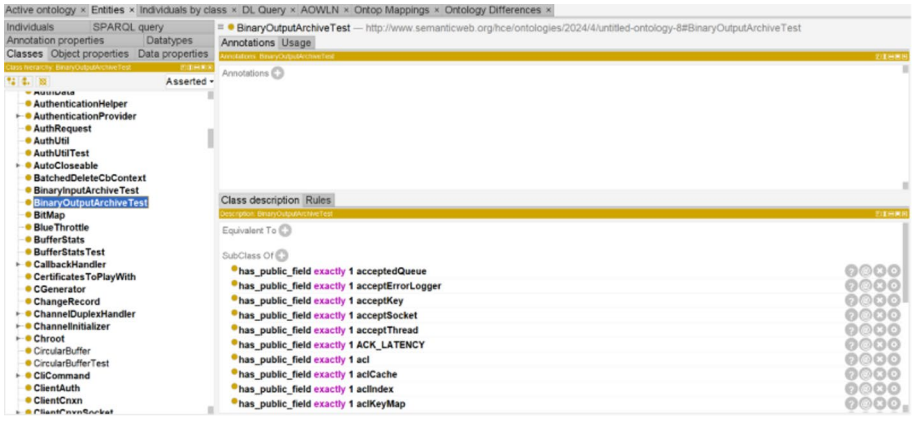


Fig. 2 The extracted ontology of commons-imaging

software repositories can be processed concurrently using distributed computing resources, while ontology reasoning tasks can be parallelized across multiple compute nodes, enabling scalable semantic analysis of industrial-scale multilingual codebases. Further, the process is organized into five main stages: (i) Ontology Extraction, (ii) Formulation of Knowledgebase, (iii) Ontology Matching, (iv) Design Smell Identification, and (v) Cross-Language Generalization, as follows:

3.3 Ontology extraction

Source code from open-source projects is reverse-engineered to extract software design artifacts (e.g., UML class diagrams), as shown in Fig. 2. These design models are then converted into ontological representations using the guidelines described in [44]. During the reverse-engineering process, the structural representation of OO software is systematically extracted by identifying software entities such as classes, inheritance hierarchies, method definitions, visibility modifiers, and dependency relationships. These language-specific structural artifacts are subsequently transformed into language-independent

ontology concepts using predefined ontology mapping rules, where classes, object properties, datatype properties, and semantic axioms provide a formal representation of the software design. Higher-level OO principles, including abstraction, encapsulation, modularity, hierarchy, and polymorphism, are then explicitly modeled through ontology axioms and SWRL reasoning rules rather than being inferred directly from programming-language syntax. Consequently, the proposed transformation preserves the conceptual semantics of software designs while minimizing semantic information loss and enabling consistent semantic reasoning across heterogeneous OO programming languages.

3.4 Formation of knowledgebase

A smell-detection knowledge base is constructed in Protégé, incorporating ontology axioms and SWRL rules that define design smells as violations of core OO principles, such as abstraction, encapsulation, modularity, and hierarchy. After the ontology structure has been established, SWRL rules are introduced to embed domain-specific logic, constraints, and inference capabilities into the model. SWRL enhances the expressive power of OWL by enabling the definition of logical rules involving OWL classes, properties, and instances. These rules capture patterns and conditions relevant to design smell detection, enabling the ontology to support automated reasoning. An illustration of this rule integration process is provided.

The knowledge representation strategy models OO design concepts as OWL classes, while inheritance, association, aggregation, dependency, encapsulation, and modularization are represented through semantic object properties. Quantitative software attributes are represented using datatype properties, whereas OO design principles are formalized as ontology axioms and SWRL rules. This semantic representation enables logical inference about conceptual software entities rather than relying on language-dependent syntactic constructs, thereby providing a reusable and extensible knowledge model for cross-language design-smell detection.

3.5 Ontology matching

After constructing the knowledge base, the next step is to compare the previously extracted ontologies with it to identify instances of design smells. This comparison is carried out through ontology matching, a process that establishes conceptual correspondences between different ontologies. It enables the identification of both structural and semantic similarities and differences between the models. To perform this task, we employed the LogMap Ontology Matching Tool [45], which facilitates the alignment of concepts, properties, and relationships across ontologies. This alignment process helps highlight deviations from core OO principles, thereby revealing potential design smells. The outcome of this mapping process, illustrated in Fig. 3, allowed us to detect inconsistencies and violations, with the results for each identified smell recorded and saved in corresponding text files.

LogMap was selected as the ontology matching technique due to its demonstrated scalability, efficient reasoning capabilities, and compatibility with OWL-based semantic knowledge bases. Unlike traditional lexical or syntactic matching approaches, LogMap combines lexical, structural, and logical reasoning techniques to generate semantically consistent ontology alignments while minimizing logical inconsistencies. Furthermore,

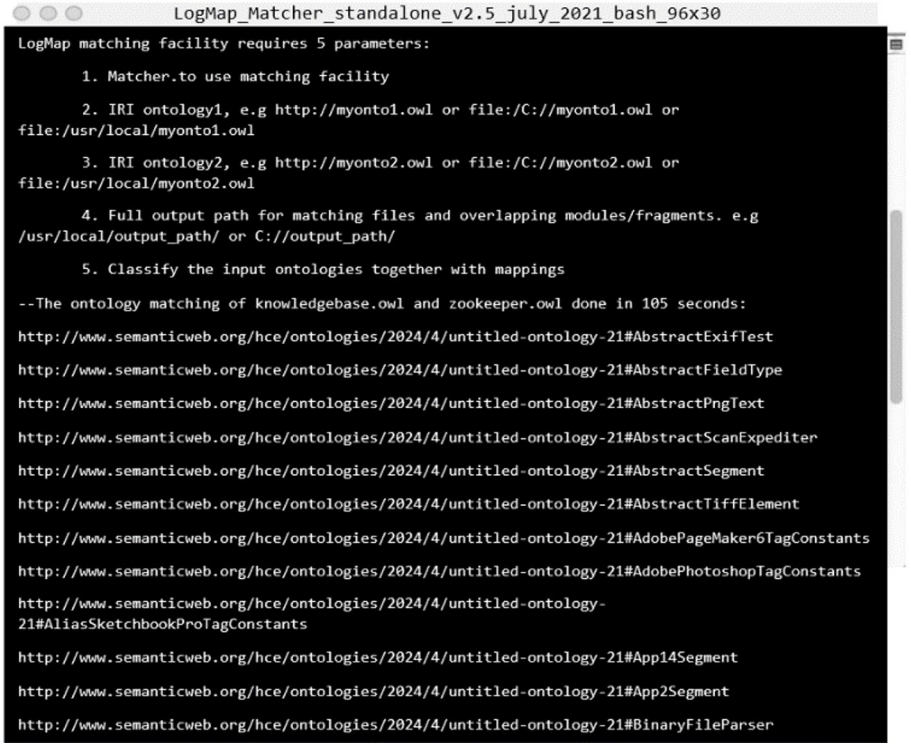


Fig. 3 Targeted smells identified in the zookeeper project

its ability to efficiently process large ontologies while preserving reasoning correctness makes it particularly suitable for multilingual software repositories and for the ontology-driven design-smell detection framework proposed in this study.

3.6 Identification of design smells

Once ontology matching is complete, the aligned elements between the extracted design models and the knowledge base are examined to identify design smells. This step involves evaluating the matched classes, relationships, and attributes to detect any breaches of core OO principles. Using LogMap's output, which includes a detailed record of each identified correspondence, we can pinpoint specific instances of design violations. These findings are then carefully reviewed and documented. A summary of the detected smells, along with their respective locations within the software design, is presented in Table 5, offering a clear view of the nature and distribution of each identified issue.

3.7 Cross-language generalization

To evaluate the cross-language generalizability of the proposed methodology, the framework was applied to a diverse set of software projects developed in Java, C++, and Python. Unlike traditional approaches that rely on language-specific syntax and code-level metrics, the proposed detection operates entirely at the design abstraction level, independent of the underlying programming language. This enables the same semantic rules to be applied uniformly across languages without the need for reconfiguration or tool adaptation. The results demonstrate that the methodology is not only language-agnostic but also effective in identifying complex design smells in systems composed of

Table 5 Frequency of design smells detected from the selected dataset

Smell type	bit-coin (c++)	Microsoft/calculator (c++)	spring-framework (java)	Elasticsearch (Java)	django (python)	flask (python)	aspnetcore (c#)	power shell (c#)	rails (ruby)	gemoji (ruby)
Multifaceted Abstraction	37	11	69	115	88	7	130	82	73	3
Unutilized Abstraction	61	17	108	157	132	9	181	127	115	5
Unnecessary Abstraction	3	1	7	10	8	1	11	7	6	0
Large Abstraction	18	6	32	54	41	3	60	39	36	1
Passive Abstraction	71	21	95	134	113	7	154	108	101	3
Duplicate Abstraction	54	13	69	101	143	11	91	78	81	6
Wide Hierarchy	3	1	5	7	5	0	8	5	4	0
Deep Hierarchy	95	22	128	165	142	9	171	135	124	4
Multipath Hierarchy	2	1	3	4	3	0	5	3	3	0
Missing Hierarchy	38	13	101	98	34	8	31	63	47	3
Cyclic Hierarchy	19	8	91	55	32	4	24	71	58	2
Deficient Encapsulation	58	19	91	138	111	6	145	118	104	3
Insufficient Modularization	20	6	28	42	34	2	45	36	32	1
Hub-Like Modularization	13	21	37	39	41	4	62	59	22	5
Broken Modularization	28	11	34	24	29	2	36	41	53	3

heterogeneous language components. This cross-language capability represents a major extension of previous work by eliminating the dependency on language-specific analysis and supporting conceptual generalization across polyglot development environments.

The proposed framework does not assume uniform implementation of OO constructs across different programming languages. Instead, source code from each language is first processed through its respective reverse-engineering mechanism to recover language-specific design information. These representations are subsequently normalized into a common semantic model comprising language-independent OO concepts such as classes, inheritance, associations, and dependencies. Consequently, semantic reasoning is performed on this unified abstraction rather than on language-specific syntax, enabling consistent design-smell detection across heterogeneous software systems.

The proposed framework is designed with a modular workflow that can be efficiently deployed in HPC environments for large-scale software analysis. The major phases of the framework, including ontology extraction, ontology matching, and semantic reasoning, can be executed independently across multiple software repositories, enabling distributed semantic processing and improved computational scalability. As the size and complexity of multilingual software systems increase, HPC infrastructures provide the computational resources required to efficiently process large ontologies and perform reasoning over extensive semantic knowledge bases. While HPC is not the primary methodological contribution of this work, it serves as an effective execution environment that enhances the scalability and practical applicability of the proposed ontology-driven framework for industrial-scale heterogeneous software systems.

The proposed ontology follows a modular knowledge engineering strategy, allowing new definitions of design smells, software quality attributes, and domain-specific reasoning rules to be incorporated without modifying the core ontology schema. This modular design supports ontology evolution while preserving semantic consistency and reasoning completeness. Furthermore, compatibility with the Open World Assumption (OWA) enables the incremental integration of concepts and relationships, ensuring the long-term extensibility and interoperability of the proposed framework across heterogeneous OO software systems.

4 Results and discussion

RQ1 What limitations arise from using language-specific code metrics in design smell detection, and how do these limitations affect the applicability of such techniques across software systems developed in heterogeneous programming languages?

Design smell detection techniques that rely on language-specific code metrics face critical limitations in their effectiveness, particularly when applied across diverse software systems written in different programming languages. These limitations stem from the fact that such metrics are inherently tied to the syntax, semantics, and structural constructs of a particular language, which restricts their reusability and generalizability. For example, tools like Designite, although capable of detecting smells in multiple languages (Java, C#, and C++), require customized parsing engines and metric calculators for each language variant [46]. This tailoring underscores that no single set of metrics applies universally; each language requires its own adaptations, leading to higher tool maintenance costs and potential inconsistencies in smell-detection logic. Moreover, smell definitions

such as "God Class" or "Feature Envy" may vary subtly across languages due to differences in language idioms, object models, and usage patterns [47, 48].

Studies have pointed out that metrics like cyclomatic complexity, coupling between objects, and lines of code behave differently across programming paradigms. For instance, researchers have emphasized that thresholds used in metric-based detection are often arbitrarily chosen and rarely portable across languages [31, 49]. What qualifies as "complex" in Java may be entirely acceptable in C++, depending on the idiomatic use of language features such as templates, multiple inheritance, or macros [50]. Furthermore, authors observed that language-specific smells may emerge due to platform-dependent features. For example, C# often encourages event-driven programming patterns that may appear as code smells when evaluated with Java-based detection criteria [51]. This results in a higher rate of anomaly detection and reduces confidence in metric-based smell-detection outcomes when applied outside their intended context.

Another significant limitation is the challenge of comparing results across codebases written in different languages. Researchers have found that metric variability across languages makes it difficult to establish uniform smell severity levels, thereby hindering the ability to conduct large-scale empirical studies on software quality [51]. Similarly, studies have reported that when design smells are examined across language boundaries, the detection precision deteriorates, leading to poor reproducibility of findings [38]. Additionally, it has been noted that tools developed for one language often fail to accommodate domain-specific architectural patterns used in others. Researchers have highlighted that design decisions in dynamic languages like Ruby or Python often violate static code metrics, even though they follow clean design principles in their idiomatic form [52]. Lastly, some authors have argued that language-specific code metrics overemphasize structural properties while ignoring contextual or behavioural nuances, which are essential for accurately identifying design flaws. This overreliance results in misleading outputs and limits the adoption of such tools in heterogeneous development environments [53]. In summary, language-specific code metrics significantly constrain the adaptability, comparability, and scalability of design smell detection tools. They create silos of detection logic that must be re-engineered for each language and fail to provide a unified view of software quality in polyglot codebases.

RQ2 Does detecting design smells at the design stage allow for a programming language-independent detection approach applicable to diverse software systems?
Yes, detecting design smells at the design stage offers a promising path to achieving a programming-language-independent detection approach. This is primarily because design-stage artifacts, such as UML diagrams, architectural models, and design specifications, capture the system's structural and behavioural intent without being tied to specific programming-language syntax. Unlike code-level smell detection methods, which rely heavily on language-specific metrics, design-stage analysis operates at a higher level of abstraction, making it inherently agnostic to the target implementation language. Research has consistently shown that many design smells, such as God Class, Cyclic Dependency, and Poor Cohesion, originate from flawed architectural or OO design decisions rather than language-specific implementation errors [48, 54]. By targeting these issues at the design phase, smell detection can transcend the limitations imposed by language syntax and compiler constraints. Early identification of design flaws significantly

enhances software maintainability and helps minimize future technical debt, particularly in complex or continuously evolving systems [55].

This has led many researchers to advocate for language-independent and semantically enriched approaches. Among these, ontology-based detection methods have gained attention for focusing on abstract representations of software design rather than relying on raw source code [10, 32]. Ontologies can capture design concepts like responsibilities, cohesion, or architectural roles at a conceptual level, independent of language syntax. These approaches not only reduce detection inaccuracies but also facilitate cross-language and design-stage detection, aligning closely with the goals of early defect prevention and sustainable software quality. In response to the limitations of metric-based techniques, some researchers have proposed semantic analyzers that reason about structural relationships and design anti-patterns in a language-agnostic manner. These approaches rely on semantic rules rather than traditional metrics, enabling more consistent detection of architectural smells across different programming languages [11, 48]. Their results demonstrated better scalability and more consistent detection across heterogeneous codebases, proving the viability of design-stage smell analysis in diverse environments.

Additional evidence suggests that developers often fail to recognize code smells during implementation. In contrast, architectural decisions made at the design stage tend to have a more lasting impact on the overall quality and maintainability of a system [52]. Furthermore, some researchers have highlighted that integrating smell detection into the early stages of development enables teams to uncover hidden quality concerns before they become code-level issues. This early intervention supports more informed and strategic design decisions [52, 56]. Several frameworks have also been proposed to operationalize design-stage detection. For example, one researcher proposed an ontology-assisted system for detecting structural flaws at the class level, enabling evaluation of design quality independently of the programming language used [20]. Likewise, one researcher applied ontology-based analysis techniques to game design systems, showcasing the ability to detect design issues across different programming languages through rule-based interpretation of design components [21].

In conclusion, moving smell detection to the design stage enables a shift from syntax-driven to semantically-aware evaluation. It supports proactive identification of design flaws in a language-agnostic manner and lays the foundation for generalizable, scalable tools and frameworks across various software platforms. To address the research question above, we applied our methodology to the selected datasets to detect design smells. The results are compiled in Table 5.

Unlike traditional techniques that rely on language-specific code metrics, our approach operates at the abstract design level using ontological models. This abstraction layer effectively eliminates the need for programming-language-specific syntax or compiler-specific constructs. Since UML models or equivalent design representations encapsulate design semantics agnostic to language, the detection of design smells becomes a semantically guided and conceptually uniform process. This supports the view that effective smell detection does not require tight coupling with the implementation language, but rather hinges on adherence to underlying OO design principles.

Our results reveal that design smells such as Multifaceted Abstraction, Deep Hierarchy, and Broken Modularization were consistently detected across datasets, regardless

of the programming language. The ability to uncover these smells depended not on language syntax but on violations of OO principles captured ontologically. By relying on a common knowledge base and formal axioms encoded in OWL and SWRL, the system was able to reason uniformly about structural and semantic inconsistencies in the design models. This reinforces the notion that early detection at the design stage is not about identifying more smells but about capturing the right smells at the right level of abstraction. In fact, our analysis shows that smells identified at this stage are often precursors to multiple code-level smells, validating that upstream quality issues can be addressed before they cascade into more complex maintenance challenges during implementation.

By decoupling the detection process from the language-specific metrics, the methodology enables a single unified detection pipeline applicable across systems written in Java, C++, Python, Ruby, and C#. This drastically reduces the overhead of reengineering detection tools for each language and provides a reusable, scalable framework for software quality analysis in heterogeneous environments. From a computational perspective, evaluating multiple large repositories simultaneously requires substantial semantic processing resources. Executing the proposed workflow within an HPC environment enables ontology extraction, reasoning, and ontology matching to be performed concurrently across multiple software projects, thereby improving scalability while preserving semantic correctness. Additionally, the results suggest that the design-level smell instances were fewer but more impactful. When compared to traditional code-level approaches, which often suffer from incorrect positive predictions or inflated detection counts due to language constructs, our ontology-based method maintained high semantic accuracy without being misled by surface-level code symptoms.

Moreover, researchers have highlighted that a clean, well-structured design reduces the likelihood of such flaws emerging later in development [46, 57, 58]. Systems with higher design complexity tend to exhibit more code smells, often at a rate of 1.5 to 3 times the number of detected design smells [29, 59]. For example, 20 design smells in a software model may translate into 30–60 code smell instances post-implementation, depending on their severity and structural impact. These challenges become particularly difficult to manage in large, OO systems due to deep inheritance chains and inter-class dependencies. By adopting a language-neutral design-level approach, tools can evaluate OO violations, such as broken abstraction, modularity, or encapsulation, without relying on language-specific syntax. This enables early reasoning over structural quality and helps contain technical debt before it spreads across the codebase. As shown in Table 5, early design-level smell detection not only simplifies refactoring but also significantly reduces the number of smells detected later in the implementation, regardless of the programming language used.

To validate our approach, we also applied the Designite tool [43] to three datasets: Java, Python and C# to compute design smells from it. Designite detects architectural, design, and code smells in source code using metrics such as method and class-level LOC, Cyclomatic Complexity (CC), Parameter Count (PC), etc. Table 6 shows the frequency of design smells detected by Designite.

Figure 4 illustrates the summary analysis generated by Designite for one of our targeted datasets. While Designite is a powerful tool for detecting code-level smells, its approach remains deeply tied to language-specific implementations. For example, to support multiple languages such as Java, Python, and C#, the developers of Designite had to build

Table 6 Frequency of design smells detected by designite

Smell type	bitcoin (c++)	Microsoft/ calculator (C++)	spring-framework (java)	elasticsearch (java)	django (python)	flask (python)	aspnetcore (c#)	power shell (c#)	rails (ruby)	gemoji (ruby)
Multifaceted Abstraction	Not supported	Not supported	84	131	95	6	142	96	Not supported	Not supported
Unutilized Abstraction			161	229	176	10	245	173		
Unnecessary Abstraction			12	17	13	1	19	14		
Large Abstraction			Not detected							
Passive Abstraction			Not detected							
Duplicate Abstraction			210	189	197	38	121	98		
Wide Hierarchy			8	11	9	1	12	9		
Deep Hierarchy			6	7	6	1	8	6		
Multipath Hierarchy			5	6	4	0	6	5		
Cyclic Hierarchy			113	67	48	7	39	79		
Missing Hierarchy			151	138	68	11	72	83		
Deficient Encapsulation			245	346	267	13	378	282		
Insufficient Modularization			146	202	159	8	217	164		
Hub-Like Modularization			59	73	69	7	83	79		
Broken Modularization			49	34	67	4	58	63		

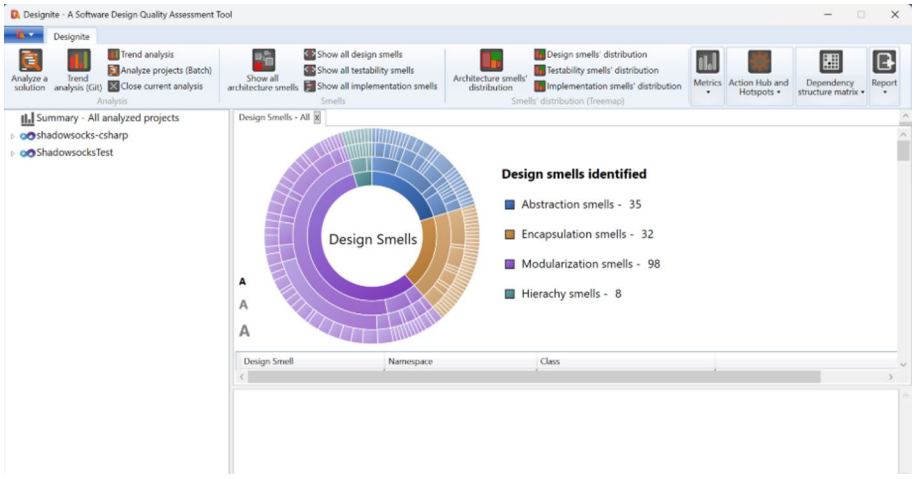


Fig. 4 Analysis summary of a project by Designite

and maintain three separate tools tailored to each language's syntax and semantics. This necessity highlights a critical limitation of code-metric-based techniques: each version must be adapted to parse and interpret different programming constructs, thereby increasing complexity and reducing generalizability. Furthermore, certain design smells, such as Large Abstraction and Passive Abstraction, remain undetected by Designite, as it relies solely on code-level metrics rather than abstract design principles.

Moreover, Designite does not support C++or Ruby for smell detection, as it only supports Java, C#, and Python. In contrast, our ontology-based methodology operates at the design stage, enabling a unified, language-agnostic framework that captures the semantics of design flaws regardless of implementation language. Instead of focusing on detecting more smells, our approach emphasizes identifying critical design issues at the right level of abstraction, before code development. This not only simplifies cross-language detection but also reduces tool maintenance overhead and improves the consistency of results across diverse software systems. By prioritizing early detection and abstraction-level analysis, our methodology offers a scalable, forward-looking alternative to traditional, language-dependent smell-detection tools.

RQ3 Can ontology-based representations outperform UML in detecting design smells at the design stage by leveraging formal semantics and reasoning capabilities?
Yes, ontology-based representations can significantly outperform UML in detecting design smells at the design stage due to their ability to encode and reason about formal semantics. While UML is widely adopted for visualizing software designs, it primarily provides syntactic and structural representations without formal semantics or embedded reasoning capabilities. As a result, UML models serve more as blueprints or communication tools rather than executable specifications for automated detection or validation of design quality. Ontologies, especially those expressed in OWL, provide a machine-understandable, semantically rich framework that supports rule-based reasoning. In the context of design smell detection, ontologies can model design constructs, such as classes, responsibilities, relationships, and modularity, as formal concepts. These concepts can then be linked to axioms representing design principles and smell definitions, enabling automated reasoning engines (like Pellet or HermiT) to identify semantic violations such as God Class, Feature Envy, or Poor Abstraction [20, 32]. In contrast,

UML cannot by itself support this type of reasoning. Although UML can be extended with the Object Constraint Language (OCL) to specify constraints, OCL's expressiveness is limited and does not match the reasoning and inference capabilities of OWL ontologies [60].

Moreover, OWL-based ontologies enable cross-model consistency checking, modularization, and integration with knowledge graphs, making them well-suited to large, complex, and evolving software systems. Several studies have validated the superiority of ontology-based detection. It has been shown that ontologies can effectively identify design smells by analyzing the semantic roles and interdependencies among design elements, an aspect often overlooked by static representations such as UML diagrams [10]. In a similar vein, an ontology-based model has been developed to represent design principles as formal axioms, allowing for the application of reasoning techniques to detect design violations more precisely than traditional model-checking approaches [10]. These findings were further supported by subsequent research demonstrating that ontology-driven detection approaches reduce incorrect classifications and significantly enhance early-stage validation of software design [12]. The comparison in Table 7 illustrates why OWL is more expressive than UML for domain modeling.

While UML has long been the de facto standard for representing software designs, its limitations in enabling reasoning, consistency checking, and semantic interpretation have become increasingly evident. Ontologies offer a powerful alternative by formalizing software design knowledge in a machine-readable format that supports automated reasoning and rule-based validation. Unlike UML, which is primarily a visual modeling tool, ontologies can explicitly define relationships, constraints, and domain rules, enabling more accurate detection of structural flaws and design smells. Ontological models also enhance reusability and interoperability across systems and programming languages by abstracting design semantics from implementation details. Researchers have highlighted these advantages, arguing that ontology-based design not only facilitates early detection of design flaws but also improves maintainability, correctness, and evolution of software system [10, 12]. Therefore, transitioning from UML to ontology-driven modeling can empower developers with tools for deeper semantic analysis and automated validation, paving the way for more robust and scalable software design practices.

Table 7 Comparison of ontologies and UML

Features	UML	OWL
Purpose	Visual modeling and documentation	Formal semantic modeling and reasoning
Semantic Expressiveness	Low – mostly syntactic	High – includes logic, axioms, and ontology semantics
Reasoning Support	None (limited through OCL constraints)	Full logical reasoning via reasoners (e.g., Pellet)
Machine Interpretability	Moderate (requires extensions)	High – designed for automated processing
Consistency Checking	Manual or tool-specific	Automated via semantic reasoners
Extensibility & Formalization	Limited (extensions through profiles)	Highly extensible through RDF, RDFS, and custom vocabularies
Support for Rule-Based Inference	Not natively supported	Fully supported via SWRL, OWL rules
Smell Detection Capability	Static, based on diagrams	Dynamic, context-aware, semantic reasoning
Cross-Domain Reusability	Limited	High ontologies can be reused across domains/systems

RQ4 Can Ontology-Based Detection Reduce False Positives in Identifying Design Smells Like God Class by Incorporating the Semantic Intent of a Class's Behavior?

Yes, ontology-based detection can significantly reduce false positives in identifying design smells like God Class by incorporating the semantic intent and behavioral roles of software components. Traditional detection techniques primarily rely on quantitative code metrics, such as method count, line count, and fan-in/fan-out values, to identify smells. While these thresholds are useful heuristics, they often fail to capture a class's contextual meaning and intended responsibility, leading to frequent misclassifications. For example, a class with numerous methods related to a cohesive concern (e.g., a DataManager class responsible for multiple well-aligned data operations) may exceed standard metric thresholds and be incorrectly labeled as a God Class. However, semantically, the class might still conform to the Single Responsibility Principle as shown in Fig. 5. Metric-based techniques overlook this nuance, leading to inflated erroneous classifications [31, 61]. Ontology-based detection addresses this shortcoming by enabling semantic modeling of class behaviors, relationships, and roles through formal ontological structures.

Using tools like Protégé, software design concepts can be defined as classes, while smells such as God Class, Low Cohesion, or Poor Abstraction can be modeled as object properties or inferred classifications. These models support rule-based reasoning, enabling systems to identify smells not only from structure but also from behavioral semantics and domain intent [20, 21]. For instance, in Fig. 5 below, the DataManager class is represented as an individual in the ontology with object properties like hasBehavior, which links it to its defined responsibilities. The system evaluates the class's role in the overall architecture. If the behaviors are unrelated or violate design axioms defined in the ontology, it may be inferred that the class exhibits the God Class smell. Conversely, if all behaviors align semantically, the smell may not be inferred, thus avoiding incorrect detections.

Several studies have supported this approach [20, 31, 61]. For instance, ontology-based modeling has demonstrated significantly lower spurious detections compared to traditional metric-based detection techniques across multiple real-world systems [10]. Further evidence indicates that formal axioms, when used to define design-smell characteristics, can be reasoned about effectively using OWL and SWRL rules, resulting in more explainable, context-sensitive detection outcomes [32]. Additionally, ontological models have demonstrated the ability to distinguish utility classes from genuine God

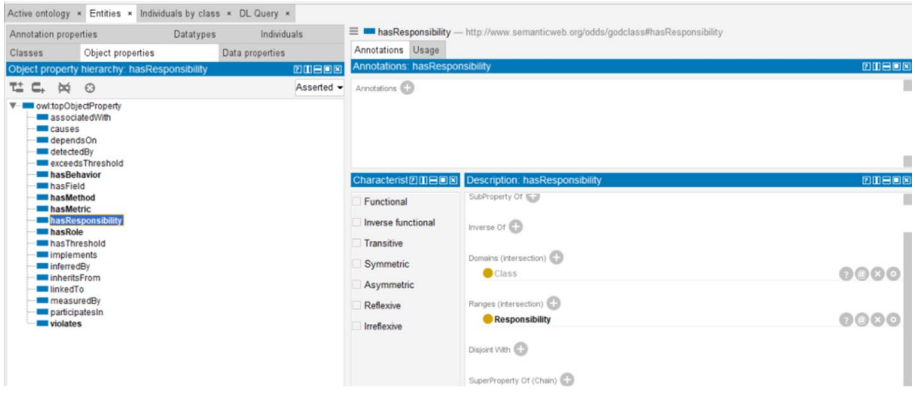


Fig. 5 Detecting false positives for god class smell

Classes by capturing the semantics of class responsibilities and behavioral patterns [9]. In conclusion, by incorporating the semantic intent and design role of classes, ontology-based approaches provide a more accurate, explainable, and context-aware method of detecting design smells like God Class. They move beyond superficial metrics to a deeper understanding of design quality, reducing misclassifications and supporting better architectural decisions early in the software development lifecycle.

In our analysis, we calculated True Positives (TPs), False Positives (FPs), and False Negatives (FNs) for each design smell type across multiple projects by comparing actual detected smells with predicted results (SPARQL Protocol and RDF Query Language (SPARQL) results). For example, in Multifaceted Abstraction, we evaluated prediction accuracy by comparing detected and predicted instances across projects. TPs were identified as the correctly predicted smells, calculated as the minimum of actual and predicted counts. FPs represented overestimated cases, defined as the difference between the actual and predicted counts when the latter was higher. FNs were computed by subtracting the predicted count from the actual count when the actual count was greater. With 89% precision and recall, resulting in an F1 score of 0.89 (see Table 8), our approach effectively captured most instances while keeping FPs low. We have shown only six projects here.

Although the proposed framework achieved high precision and recall across the evaluated projects, a limited number of incorrect detections were observed. These cases primarily involved utility classes with numerous related operations, framework-generated abstractions, and classes exhibiting complex dependency structures that closely resembled legitimate design smell patterns. In such situations, structural similarities occasionally resulted in ambiguous semantic interpretations, leading to isolated incorrect detections. Nevertheless, these cases were infrequent and did not substantially affect the overall performance of the proposed ontology-driven framework, which consistently maintained high semantic accuracy across multilingual software systems.

The results were aggregated to calculate overall metrics: precision was 0.91, indicating the ratio of correctly predicted positive observations to the total predicted positives. Recall was 0.92, showing the ratio of correctly predicted positive observations to all observations in the actual class. The F1 Score, which balances precision and recall, was 0.92. These metrics suggest the model performed exceptionally well in identifying design smells across the projects, with high accuracy and completeness in its predictions.

5 Threats to validity

The proposed framework is subject to several potential threats to validity. Construct validity concerns whether the ontology and semantic reasoning rules accurately represent OO design principles and design smell definitions. To reduce this threat, the ontology was developed from established OO concepts and validated through OWL axioms and SWRL reasoning rules. Nevertheless, emerging design paradigms or newly identified design smells may require future extensions to the ontology.

Internal validity: relates to the reverse-engineering and ontology engineering processes used to transform source code into semantic representations. Although language-specific syntactic constructs are abstracted into language-independent OO concepts, certain implementation-specific characteristics may not be represented explicitly. This limitation is mitigated by preserving core structural relationships, including classes,

inheritance, associations, dependencies, and visibility modifiers, before semantic reasoning is performed.

External validity: concerns the generalizability of the proposed framework. The evaluation was conducted on ten actively maintained open-source projects implemented in Java, C++, C#, Python, and Ruby, representing diverse application domains and software sizes. Although these projects provide a representative benchmark for multilingual OO software systems, additional validation across industrial software repositories, larger datasets, and emerging OO programming languages would strengthen the proposed framework's general applicability.

6 Conclusion and future direction

This study presents a novel ontology-based approach for detecting design smells at the design stage, overcoming the limitations of traditional metric-based techniques that are often tightly coupled to specific programming languages. By abstracting design semantics through formal ontologies and incorporating reasoning mechanisms, our method enables early, accurate, and context-aware identification of design anomalies, most notably the God Class smell. Unlike code-level tools that rely on structural thresholds and language-specific parsers, the proposed framework leverages domain knowledge and behavioral intent to reduce erroneous smell identifications and enhance cross-language generalizability. The experimental results, supported by reverse-engineered ontologies across multiple OO programming languages, validate the effectiveness and adaptability of our approach.

Beyond multilingual design smell detection, this work demonstrates the potential of semantic knowledge engineering as a foundation for intelligent software quality assurance. The proposed ontology-driven framework provides a reusable semantic infrastructure that can evolve with emerging software architectures while remaining suitable for deployment within HPC environments. Future research will investigate distributed ontology reasoning, automated ontology evolution, and semantic knowledge graphs to enable scalable software maintenance, continuous quality assessment, and intelligent decision support for large industrial software ecosystems.

Table 8 Precision, recall and F1 measure of the examined application

Design Smell	Bitcoin			Calculator			SPRING-FRAMEWORK			ELASTICSEARCH		
	Precision	Recall	F1 Measure	Precision	Recall	F1 Measure	Precision	Recall	F1 Measure	Precision	Recall	F1 Measure
Multifaceted Abstraction	(33/33+4)=89%	(33/33+4)=89%	0.89	(10/10+1)=91%	(10/10+1)=91%	0.91	(62/62+7)=90%	(62/62+6)=91%	0.90	(103/103+12)=90%	(103/103+11)=90%	0.90
Unutilized Abstraction	(55/55+6)=90%	(55/55+6)=90%	0.90	(15/15+2)=88%	(15/15+2)=88%	0.88	(97/97+11)=90%	(97/97+10)=91%	0.90	(141/141+16)=89%	(141/141+15)=90%	0.89
Unnecessary Abstraction	(3/3+0)=100%	(3/3+0)=100%	1.00	(1/1+0)=100%	(1/1+0)=100%	1.00	(6/6+1)=86%	(6/6+1)=86%	0.86	(9/9+1)=90%	(9/9+1)=90%	0.90
Large Abstraction	(16/16+2)=89%	(16/16+2)=89%	0.89	(5/5+1)=83%	(5/5+1)=83%	0.83	(29/29+3)=91%	(29/29+3)=91%	0.91	(49/49+5)=91%	(49/49+5)=91%	0.91
Passive Abstraction	(64/64+7)=90%	(64/64+7)=90%	0.90	(19/19+2)=90%	(19/19+2)=90%	0.90	(86/86+9)=91%	(86/86+9)=91%	0.91	(121/121+13)=90%	(121/121+13)=90%	0.90
Duplicate Abstraction	(48/48+6)=89%	(48/48+6)=89%	0.89	(14/14+2)=88%	(14/14+2)=88%	0.88	(91/91+10)=90%	(91/91+9)=91%	0.90	(152/152+17)=90%	(152/152+15)=91%	0.90
Wide Hierarchy	(3/3+0)=100%	(3/3+0)=100%	1.00	(1/1+0)=100%	(1/1+0)=100%	1.00	(5/5+0)=100%	(5/5+0)=100%	1.00	(7/7+0)=100%	(7/7+0)=100%	1.00
Deep Hierarchy	(90/90+5)=95%	(90/90+5)=95%	0.95	(21/21+1)=95%	(21/21+1)=95%	0.95	(122/122+6)=95%	(122/122+6)=95%	0.95	(157/157+8)=95%	(157/157+8)=95%	0.95
Multipath Hierarchy	(2/2+0)=100%	(2/2+0)=100%	1.00	(1/1+0)=100%	(1/1+0)=100%	1.00	(3/3+0)=100%	(3/3+0)=100%	1.00	(4/4+0)=100%	(4/4+0)=100%	1.00
Missing Hierarchy	(36/36+2)=95%	(36/36+2)=95%	0.95	(10/10+1)=91%	(10/10+1)=91%	0.91	(67/67+4)=94%	(67/67+4)=94%	0.94	(113/113+6)=95%	(113/113+6)=95%	0.95
Cyclic Hierarchy	(18/18+1)=95%	(18/18+1)=95%	0.95	(6/6+0)=100%	(6/6+0)=100%	1.00	(33/33+2)=94%	(33/33+2)=94%	0.94	(55/55+3)=95%	(55/55+3)=95%	0.95
Deficient Encapsulation	(53/53+5)=91%	(53/53+5)=91%	0.91	(17/17+2)=89%	(17/17+2)=89%	0.89	(82/82+9)=90%	(82/82+9)=90%	0.90	(127/127+11)=92%	(127/127+11)=92%	0.92
Insufficient Modularization	(18/18+2)=90%	(18/18+2)=90%	0.90	(6/6+0)=100%	(6/6+0)=100%	1.00	(25/25+3)=89%	(25/25+3)=89%	0.89	(39/39+3)=93%	(39/39+3)=93%	0.93
Hub-Like Modularization	(12/12+1)=92%	(12/12+1)=92%	0.92	(4/4+0)=100%	(4/4+0)=100%	1.00	(22/22+2)=92%	(22/22+2)=92%	0.92	(37/37+3)=92%	(37/37+3)=92%	0.92
Broken Modularization	(26/26+2)=93%	(26/26+2)=93%	0.93	(7/7+1)=88%	(7/7+1)=88%	0.88	(47/47+5)=90%	(47/47+5)=90%	0.90	(80/80+7)=92%	(80/80+7)=92%	0.92
Average	92%	92.1%	0.92	92%	92.1%	0.92	93%	93%	0.93	93%	93%	0.93
	DJANGO			**Flask**			**ASPNETCORE**			**PowerShell**		
Design Smell	Precision	Recall	F1 Measure	Precision	Recall	F1 Measure	Precision	Recall	F1 Measure	Precision	Recall	F1 Measure
Multifaceted Abstraction	(79/79+9)=90%	(79/79+8)=91%	0.90	(22/22+3)=88%	(9/9+1)=90%	0.89	(117/117+13)=90%	(117/117+12)=91%	0.90	(74/74+8)=90%	(74/74+7)=91%	0.90
Unutilized Abstraction	(119/119+13)=90%	(119/119+12)=91%	0.90	(22/22+3)=88%	(9/9+1)=90%	0.89	(163/163+18)=90%	(163/163+16)=91%	0.91	(114/114+13)=90%	(114/114+12)=91%	0.90
Unnecessary Abstraction	(7/7+1)=88%	(7/7+1)=88%	0.88	(22/22+3)=88%	(9/9+1)=90%	0.89	(10/10+1)=91%	(10/10+1)=91%	0.91	(6/6+1)=86%	(6/6+1)=86%	0.86
Large Abstraction	(37/37+4)=90%	(37/37+4)=90%	0.90	(22/22+3)=88%	(9/9+1)=90%	0.89	(54/54+6)=90%	(54/54+5)=92%	0.91	(35/35+4)=90%	(35/35+4)=90%	0.90
Passive Abstraction	(102/102+11)=90%	(102/102+10)=91%	0.90	(22/22+3)=88%	(9/9+1)=90%	0.89	(139/139+15)=90%	(139/139+14)=91%	0.91	(97/97+11)=90%	(97/97+10)=91%	0.90
Duplicate Abstraction	(116/116+13)=90%	(116/116+12)=91%	0.90	(22/22+3)=88%	(9/9+1)=90%	0.89	(82/82+9)=90%	(82/82+8)=91%	0.91	(86/86+10)=90%	(86/86+9)=91%	0.90
Wide Hierarchy	(5/5+0)=100%	(5/5+0)=100%	1.00	(19/19+1)=95%	(47/47+3)=94%	0.94	(8/8+0)=100%	(8/8+0)=100%	1.00	(5/5+0)=100%	(5/5+0)=100%	1.00
Deep Hierarchy	(135/135+7)=95%	(135/135+14)=91%	0.93	(19/19+1)=95%	(47/47+3)=94%	0.94	(162/162+9)=95%	(162/162+10)=94%	0.94	(128/128+7)=95%	(128/128+7)=95%	0.95
Multipath Hierarchy	(3/3+0)=100%	(3/3+0)=100%	1.00	(19/19+1)=95%	(47/47+3)=94%	0.94	(5/5+0)=100%	(5/5+0)=100%	1.00	(3/3+0)=100%	(3/3+0)=100%	1.00
Missing Hierarchy	(82/82+9)=90%	(82/82+8)=91%	0.90	(19/19+1)=95%	(47/47+3)=94%	0.94	(29/29+2)=94%	(29/29+2)=94%	0.94	(50/50+3)=94%	(50/50+3)=94%	0.94
Cyclic Hierarchy	(41/41+4)=91%	(41/41+4)=91%	0.91	(19/19+1)=95%	(47/47+3)=94%	0.94	(23/23+1)=96%	(23/23+1)=96%	0.96	(25/25+1)=96%	(25/25+1)=96%	0.96
Deficient Encapsulation	(102/102+9)=92%	(102/102+9)=92%	0.92	(23/23+2)=92%	(28/28+2)=93%	0.92	(133/133+12)=92%	(133/133+11)=92%	0.92	(108/108+10)=92%	(108/108+10)=92%	0.92
Insufficient Modularization	(31/31+3)=91%	(31/31+3)=91%	0.91	(23/23+2)=92%	(28/28+2)=93%	0.92	(41/41+4)=91%	(41/41+3)=93%	0.92	(33/33+3)=92%	(33/33+3)=92%	0.92

Table 8 (continued)

Design Smell	Bitcoin			Calculator			SPRING-FRAMEWORK			ELASTICSEARCH		
	Precision	Recall	F1 Measure	Precision	Recall	F1 Measure	Precision	Recall	F1 Measure	Precision	Recall	F1 Measure
Hub-Like Modularization	(28/28 + 3) = 90%	(28/28 + 3) = 90%	0.90	(23/23 + 2) = 92%	(28/28 + 2) = 93%	0.92	(57/57 + 5) = 92%	(57/57 + 5) = 92%	0.92	(17/17 + 2) = 89%	(17/17 + 2) = 89%	0.89
Broken Modularization	(59/59 + 7) = 89%	(59/59 + 6) = 91%	0.90	(23/23 + 2) = 92%	(28/28 + 2) = 93%	0.92	(33/33 + 3) = 92%	(33/33 + 3) = 92%	0.92	(40/40 + 4) = 91%	(40/40 + 4) = 91%	0.91
Average	92%	92.1%	0.92	92%	92.1%	0.92	93%	93%	0.93	93%	93%	0.93

Author contributions
All authors (Sabiyyah Sabir, Ghulam Rasool, Tariq Umer, Harun Elkiran, and Jawad Rasheed) contributed equally to each section (from conceptualization to Writing - original draft, review, and editing). All authors have read and agreed to the published version of the manuscript.

Funding
The authors declare that this research received no external funding.

Data availability
The datasets used and/or analyzed during the current study are available from the author (Ghulam Rasool; grasool@cuilahore.edu.pk) on reasonable request.

Declarations

Ethical approval
Not applicable.

Consent to participate
Not applicable.

Consent for publication
Not applicable.

Competing interests
The authors declare no competing interests.

Received: 12 July 2026 / Accepted: 4 September 2026



References

1. Samarthyam G, Suryanarayana G, Sharma T. Refactoring for Software Architecture Smells, in Proceedings of the 1st International Workshop on Software Refactoring, in IWoR 2016. New York, NY, USA: Association for Computing Machinery, 2016, pp. 1–4. https://doi.org/10.1145/2975945.2975946.
2. Rahman A, Sharma T. Lessons from research to practice on writing better quality puppet scripts, in 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2022;63–67. https://doi.org/10.1109/SANER53432.2022.00019.
3. Sharma T, Singh P, Spinellis D. An empirical investigation on the relationship between design and architecture smells. Empir Softw Eng. 2020;25(5):4020–68. https://doi.org/10.1007/s10664-020-09847-2.
4. AbuHassan A, Alshayeb M, Ghouti L. Software smell detection techniques: A systematic literature review. J Softw Evol Process. 2021;33(3):e2320. https://doi.org/10.1002/smr.2320.
5. Suryanarayana G, Samarthyam G, Sharma T. Chapter 2 - Design Smells. In: Suryanarayana G, Samarthyam G, Sharma T, editors. in Refactoring for Software Design Smells. Boston: Morgan Kaufmann; 2015. pp. 9–19. https://doi.org/10.1016/B978-0-12-801397-7.00002-3.
6. Pereira dos J, Reis F, Brito e Abreu G, de Figueiredo, Carneiro, Anslow C. Code Smells Detection and Visualization: A Systematic Literature Review. Arch Comput Methods Eng. 2022;29(1):47–94. https://doi.org/10.1007/s11831-021-09566-x.
7. Luo Y, Hoss A, Carver DL. An ontological identification of relationships between anti-patterns and code smells, in 2010 IEEE Aerospace Conference, 2010;1–10. https://doi.org/10.1109/AERO.2010.5446811.
8. Roldan-Molina GR, Mendez JR, Yevseyeva I, Basto-Fernandes V. Ontology fixing by using software engineering technology. Appl Sci. 2020;10(18). https://doi.org/10.3390/app10186328.
9. da Silva Carvalho LP, Novais R, Salvador L, de Neto MG. An ontology-based approach to analyzing the occurrence of code smells in software, In International Conference on Enterprise Information Systems, 2017;155–165.
10. Castellano IL, et al. An Ontology-Based Approach to Reduce the Negative Impact of Code Smells in Software Development Projects. IEEE Access. 2023;11:100146–53. https://doi.org/10.1109/ACCESS.2023.3300575.
11. Chondamrongkul N, Sun J, Warren I, Lee SUJ. Semantic-based architecture smell analysis. In: Proceedings of the 8th International Conference on Formal Methods in Software Engineering (FormaliSE '20). New York (NY): Association for Computing Machinery; 2020. p. 109-118. https://doi.org/10.1145/3372020.3391564.
12. Sabir S, Rasool G. A lightweight approach for detection of software design smells. In: 2025 6th International Conference on Advancements in Computational Sciences (ICACS). 2025. p. 1-4. https://doi.org/10.1109/ICACS64902.2025.10937837
13. Silva VJS, Dorça FA. An automatic and intelligent approach for supporting teaching and learning of software engineering considering design smells in object-oriented programming. In: 2019 IEEE 19th International Conference on Advanced Learning Technologies (ICALT). 2019. p. 321-323. https://doi.org/10.1109/ICALT.2019.00100
14. Mekruksavanich S. Identifying behavioral design flaws in evolving object-oriented software using an ontology-based approach. In: 2017 13th International Conference on Signal-Image Technology & Internet-Based Systems (SITIS). 2017. p. 424-429. https://doi.org/10.1109/SITIS.2017.76.
15. Mejhed Mkhinini M, Labbani-Narsis O, Nicolle C. Combining UML and ontology: An exploratory survey. Comput Sci Rev. 2020;35:100223. https://doi.org/10.1016/j.cosrev.2019.100223.
16. Abdalazeim A, Meziane F. A review of the generation of requirements specification in natural language using objects UML models and domain ontology. Procedia Comput Sci. 2021;189:328–34. https://doi.org/10.1016/j.procs.2021.05.102.
17. Guizzardi G, Guarino N. Explanation, semantics, and ontology. Data Knowl Eng. 2024;153:102325. https://doi.org/10.1016/j.datak.2024.102325.
18. Guizzardi G, Guarino N. Semantics, Ontology and explanation, 2023. [Online]. Available: https://arxiv.org/abs/2304.11124.

19. Veizaga A, Shin SY, Briand LC. Automated Smell Detection and Recommendation in Natural Language Requirements. IEEE Trans Softw Eng. 2024;50(4):695–720. https://doi.org/10.1109/TSE.2024.3361033.
20. Mekruksavanich S. Ontology-Assisted Structural Design Flaw Detection of Object-Oriented Software. In: Theeramunkong T, Kongkachandra R, Ketcham M, Hnoohom N, Songmuang P, Supnithi T, Hashimoto K, editors. in Advances in Intelligent Informatics, Smart Technology and Natural Language Processing. Cham: Springer International Publishing; 2019. pp. 119–28.
21. Haendler T, Neumann G. Ontology-based analysis of game designs for software refactoring., in CSEDU (1), 2019;24–35.
22. Joshua NM, Garcia D, Popescu G, Edwards. Toward a catalogue of architectural bad smells, in In International conference on the quality of software architectures, Berlin: Springer, 2009;146–162.
23. Garcia J, Popescu D, Edwards G, Medvidovic N. Toward a Catalogue of Architectural Bad Smells. In: Mirandola R, Gorton I, Hofmeister C, editors. in Architectures for Adaptive Software Systems. Berlin, Heidelberg: Springer Berlin Heidelberg; 2009. pp. 146–62.
24. Ganesh SG, Sharma T, Suryanarayana G. Towards a Principle-based Classification of Structural Design Smells. J Object Technol. 2013;12(2):1.
25. Fowler M. Refactoring: improving the design of existing code. Addison-Wesley Professional. 2018.
26. Gentili E, Falessi D. Characterizing Requirements Smells. In: Kadgien R, Jedlitschka A, Janes A, Lenarduzzi V, Li X, editors. in Product-Focused Software Process Improvement. Cham: Springer Nature Switzerland; 2024. pp. 387–98.
27. Kasenchak B, Lehnert A, Loh G. Use Case: Ontologies and RDF-Star for Knowledge Management. In: Verborgh R, Dimou A, Hogan A, d'Amato C, Tiddi I, Bröring A, Mayer S, Ongenae F, Tommasini R, Alam M, editors. in The Semantic Web: ESWC 2021 Satellite Events. Cham: Springer International Publishing; 2021. pp. 254–60.
28. Betancourt IC, Martínez NS, García MN, Grass OR, Vázquez OC. Exploiting an Ontology-Based Solution to Study Code Smells. In: Singh PK, Veselov G, Vyatkin V, Pljonkin A, Dodero JM, Kumar Y, editors. in Futuristic Trends in Network and Communication Technologies. Eds., Singapore: Springer Singapore; 2021. pp. 234–46.
29. Alkharabsheh K, Crespo Y, Manso E, Taboada JA. Software Design Smell Detection: a systematic mapping study. Softw Qual J. 2019;27(3):1069–148. https://doi.org/10.1007/s11219-018-9424-8.
30. Tsantalis N, Chaikalis T, Chatzigeorgiou A. JDeodorant: identification and removal of type-checking bad smells, in 2008 12th European Conference on Software Maintenance and Reengineering, 2008, pp. 329–331. https://doi.org/10.1109/CSMR.2008.4493342.
31. Moha N, Gueheneuc Y-G, Duchien L, Le Meur A-F. DECOR: A Method for the Specification and Detection of Code and Design Smells. IEEE Trans Softw Eng. 2010;36(1):20–36. https://doi.org/10.1109/TSE.2009.50.
32. Sabir S, Rasool G, Umer T, Kraiem N, Nam Y. An Ontology-Enabled Approach for the Detection of Software Design Smells. IEEE Access. 2025;13:129056–73. https://doi.org/10.1109/ACCESS.2025.3590267.
33. Arcelli Fontana F, Mantyla MV, Zanoni M, Marino A. Comparing and Experimenting Machine Learning Techniques for Code Smell Detection. Empir Softw Eng. 2016;21(3):1143–91. https://doi.org/10.1007/s10664-015-9378-4.
34. Palomba F, Bavota G, Di Penta M, Oliveto R, De Lucia A. Detecting Bad Smells in Source Code Using Change History Information. IEEE Trans Softw Eng. 2018;44(11):1069–96. https://doi.org/10.1109/TSE.2017.2696978.
35. Khomh F, Penta MD, Guéhéneuc YG. An exploratory study of the impact of code smells on software change-proneness. In: Working Conference on Reverse Engineering. IEEE; 2009. p. 75-84. https://doi.org/10.1109/WCRE.2009.28.
36. Sajnani H, Saini V, Svajlenko J, Roy CK, Lopes C. SourcererCC: scaling code clone detection to big code. In: Proceedings of the 38th International Conference on Software Engineering. New York (NY): Association for Computing Machinery; 2016. p. 1157-1168. https://doi.org/10.1145/2884781.2884877.
37. Bavota G, De Lucia A, Penta MD, Oliveto R. The Impact of Code Smells on Software Fault-Proneness. Empir Softw Eng. 2015;20(4):1018–53. https://doi.org/10.1007/s10664-014-9346-5.
38. Palomba F, Bavota G, Di Penta M, Oliveto R, De Lucia A. Do they really smell bad? A study on developers' perception of bad code smells. In: 2014 IEEE International Conference on Software Maintenance and Evolution. 2014. p. 101-110. https://doi.org/10.1109/ICSME.2014.32.
39. Yamashita A, Moonen L. Do code smells reflect important maintainability aspects? In: IEEE International Conference on Software Maintenance. IEEE; 2013. p. 306-315. https://doi.org/10.1109/ICSM.2013.43.
40. Liu H, Xia X, Lo D, Li S, Grundy J. Deep learning based code smell detection. In: IEEE International Conference on Software Maintenance and Evolution. IEEE; 2019. p. 217-227. https://doi.org/10.1109/ICSME.2019.00049
41. Lima R, et al. Exploring Transfer learning for multilingual software quality: code smells, bugs, and harmful. J Softw Eng Res Dev. 2025;13(1). https://doi.org/10.5753/jserd.2025.4593.
42. Li Z, et al. Automated Detection of Inter-Language Design Smells in Multi-Language Deep Learning Frameworks. Inf Softw Technol. 2025;177:107656. https://doi.org/10.1016/j.infsof.2024.107656.
43. Ali I, Rizvi SSH, Adil SH. Enhancing Software Quality with AI: A Transformer-Based Approach for Code Smell Detection. Appl Sci. 2025;15(8):4559. https://doi.org/10.3390/app15084559.
44. Xu Z, Ni Y, He W, Lin L, Yan Q. Automatic extraction of OWL ontologies from UML class diagrams: a semantics-preserving approach. World Wide Web. 2012;15(5):517–45. https://doi.org/10.1007/s11280-011-0147-z.
45. Jiménez-Ruiz E, Cuenca Grau B. LogMap: Logic-Based and Scalable Ontology Matching. In: Aroyo L, Welty C, Alani H, Taylor J, Bernstein A, Kagal L, Noy N, Blomqvist E, editors. in The Semantic Web -- ISWC 2011. Berlin, Heidelberg: Springer Berlin Heidelberg; 2011. pp. 273–88.
46. Sharma T, Mishra P, Tiwari R. Designite: a software design quality assessment tool. In: Proceedings of the 1st International Workshop on Bringing Architectural Design Thinking into Developers' Daily Activities (BRIDGE '16). New York (NY): Association for Computing Machinery; 2016. p. 1-4. https://doi.org/10.1145/2896935.2896938.
47. Zhang C, Budgen D. What Do We Know about the Effectiveness of Software Design Patterns? IEEE Trans Softw Eng. 2012;38(5):1213–31. https://doi.org/10.1109/TSE.2011.79.
48. Arcoverde R, Garcia A, Figueiredo E. Understanding the longevity of code smells: preliminary results of an explanatory survey. In: Proceedings of the 4th Workshop on Refactoring Tools (WRT '11). New York (NY): Association for Computing Machinery; 2011. p. 33-36. https://doi.org/10.1145/1984732.1984740.
49. Tsantalis N, Chatzigeorgiou A, Stephanides G, Halkidis ST. Design Pattern Detection Using Similarity Scoring. IEEE Trans Softw Eng. 2006;32(11):896–909. https://doi.org/10.1109/TSE.2006.112.

50. Lanza M, Marinescu R. Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems. Berlin: Springer; 2007.
51. Khomh F, Gueheneuc Y-G. Do Design patterns impact software quality positively? in 2008 12th European Conference on Software Maintenance and Reengineering, 2008;274–278. https://doi.org/10.1109/CSMR.2008.4493325.
52. Yamashita A, Moonen L. Do developers care about code smells? An exploratory survey, in 2013 20th Working Conference on Reverse Engineering (WCRE), 2013;242–251. https://doi.org/10.1109/WCRE.2013.6671299.
53. Ganesh GSSG, Sharma T. Towards a Principle-based Classification of Structural Design Smells. J Object Technol. 2013;12(2):1–29.
54. Sharma H, Kalia AK, A., Singh. Metrics identification for measuring object oriented software quality. nternational J soft Comput Eng. 2012;2(5):255–8.
55. Palomba F, Bavota G, Di Penta M, Fasano F, Oliveto R, De Lucia A. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation, in Proceedings of the 40th International Conference on Software Engineering, in ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018; 482. https://doi.org/10.1145/3180155.3182532.
56. 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. 2015;41(5):462–89. https://doi.org/10.1109/TSE.2014.2372760.
57. Reyaz S, Ranjan D. Study of various quality metrics suitable for the object oriented environment.
58. Vidal SA, Marcos C, Díaz-Pace JA. An approach to prioritize code smells for refactoring. Autom Softw Eng. 2016;23(3):501–32. https://doi.org/10.1007/s10515-014-0175-x.
59. Peter Kokol SZ, Kokol M. Code smells: a synthetic narrative review, arXiv. 2103; 01088 https://doi.org/10.48550/arXiv.2103.01088.
60. Geisler S, Quix C, Weber S, Jarke M. Ontology-based data quality management for data streams. J Data Inf Qual. Oct. 2016;7(4). https://doi.org/10.1145/2968332.
61. Fontana FA, Ferme V, Marino A, Walter B, Martenka P. Investigating the impact of code smells on system's quality: an empirical study on systems of different application domains, in 2013 IEEE International Conference on Software Maintenance, 2013;260–269. https://doi.org/10.1109/ICSM.2013.37.

Publisher's note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.