Information and Software Technology 184 (2025) 107760

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

SmellDSL: A domain-specific language to assist developers in specifying code smell patterns

Robson Keemps [a,b,*], Kleinner Farias [a], Rafael Kunst [a], Carlos Carbonera [a], Willian Bolzan [a,c]

[a] PPGCA, University of Vale do Rio dos Sinos, Av. Unisinos, 950, São Leopoldo, RS, Brazil
[b] Federal Institute of Mato Grosso, Avenida Sen. Filinto Müller, 953, Cuiabá, MT, Brazil
[c] Federal Institute of Santa Catarina, Rua Deputado Olices Pedra de Caldas, 480, Tubarão, SC, Brazil

ARTICLE INFO

Keywords:
Architectural problem
Code smells
Code smell specification
Domain-specific language
Empirical study

ABSTRACT

Context: The current literature has widely investigated *code smell patterns* over the years, which describe specific source code characteristics that indicate potential problems or areas for improvements. Empirical studies suggest that (i) metric-based strategies for code smell detection are not effective and overload the developers with false positives; (ii) code smell specifications are informal, ambiguous, and not supported by traditional IDEs like Eclipse platform; and (iii) the identification of code smells depends on the perception of software development teams.
Objective: This article, therefore, proposes SmellDSL, a tool-supported domain-specific language to assist developers when specifying code smell patterns. SmellDSL benefits developers by introducing Eclipse built-in constructs that enable the specification of team-sensitive code smell patterns. Developers can write rules to specify single or composite architectural problems (*e.g., Misplaced Concerns*) and suggest code refactorings regarding severe architectural degradation symptoms.
Method: We conducted an empirical study with 35 developers who specified eight code smells using SmellDSL, generating 280 evaluation scenarios.
Results: The main results, supported by statistical tests, suggest that SmellDSL requires low effort to specify code smell patterns and promotes a high rate of correctly code smell specifications.
Conclusion: We contribute with a domain-specific language for the specification of code smell patterns, empirical evidence on its usefulness, and draw worth-investigating research challenges by the research community.

Contents

* Corresponding author at: Federal Institute of Mato Grosso, Avenida Sen. Filinto Müller, 953, Cuiabá, MT, Brazil.
E-mail address: robson.keemps@edu.unisinos.br (R. Keemps).

https://doi.org/10.1016/j.infsof.2025.107760
Received 2 July 2024; Received in revised form 13 February 2025; Accepted 17 April 2025
Available online 3 May 2025

1. Introduction

Code smells are internal source code structures that challenge design principles or rules, negatively impacting the internal quality of evolving software systems [1,2]. Code smells typically arise when developers carelessly change the source code to add new features or even perform maintenance tasks [3–7]. The inappropriate changes have been investigated over the decades and recognized as indicators of potential software design problems [2,6,8,9]. Recent studies [5,6,10] present reflections and insightful refactoring strategies on how to mitigate such design problems. As the software systems evolve, the agglomeration of code smells can cause the architecture to diverge from what is desired. Empirical studies [8,11–13] suggest that metric-based strategies for code smell detection remain ineffective and overload software developers with several false positives [14,15]; code smell specifications are informal, ambiguous and not supported by traditional IDEs like Eclipse platform [16]; and the identification of code smells depend on the perception of software development teams. Defining what constitutes a code smells (and what does not) remains a subjective task. Developers need to analyze several code units (*e.g.*, classes, interfaces, and packages), which is a time-consuming and challenging task.

Some recent studies [8,11–13] used software design metrics (with threshold values) to identify code smells, which are typically defined and specified informally, making understanding and refactoring tasks difficult. The code smell definition often contains "noises" (*e.g.*, unnecessary or ambiguous definitions of code units and relationships), which are generally of no interest to the design problems analysis and thus contribute to the lack of precision of the code smell specifications. For example, classes in Java programs with many fields, methods, or lines of source code are considered a Large Class code smell. Consequently, understanding and specifying a bad smell becomes a subjective and error-prone task. The current literature still lacks approaches that help development teams specify such code smells more rigorously. Even worse, previous study findings [8,17] reveal a significant lack of consensus among developers about how to detect a widely-known catalog of code smells [2], highlighting the subjective nature of this process. Still, factors such as background and experience do not consistently influence agreement levels. Instead, the identification of code smells is strongly influenced by the personal heuristics employed by individual developers [17]. This subjectivity highlights the challenge of achieving a universal standard of code smells (and their detection), as perceptions of what a bad smell is vary across projects and among developers. In this sense, considering developers' perceptions concerning bad smells is essential since they can differ when multiple developers evaluate the same source code [8,17]. This means software developers assess code smells' presence (or absence), considering divergent code smell detection heuristics. Consequently, code quality levels within a project can vary significantly, with different pieces of code exhibiting various levels of quality [8].

This article, therefore, proposes SmellDSL, a tool-supported domain-specific language to assist developers when specifying code smell patterns. SmellDSL differentiates itself by supporting developer-sensitive code smell design, employing concepts from object-oriented languages, and representing detection heuristics through design problem rules. The language provides developers with constructs (*e.g.*, `smelltype`, `smell`, `feature`, `symptom`, `treatment` and `rule`) to specify code smells. Tool support (namely the SmellDSL tool) was implemented as an Eclipse Platform plug-in. We conducted an empirical study with 35 participants who specified 8 code smells using the SmellDSL tool, generating 280 evaluation scenarios. This evaluation allowed us to grasp the effect of SmellDSL from the viewpoint of software developers in terms of specification effort, correctness, and error rate. Our results, supported by statistical tests, suggest that, when using SmellDSL, software developers invest little effort in specifying code smells (less than 10 min) and produce a high number of correct specifications (above 80%). Furthermore, incorrect specifications have a low error rate (below 20%). Participants were evaluated individually for each proposed task.

The contributions of this paper are (1) a domain-specific language designed to support developer-sensitive smell specification by leveraging object-oriented concepts to enhance familiarity and ease of use, implementing detection heuristics through explicit design problem rules, and enabling precise, developer-focused code smell definitions. Together, these features improve code smell specification's accuracy (high

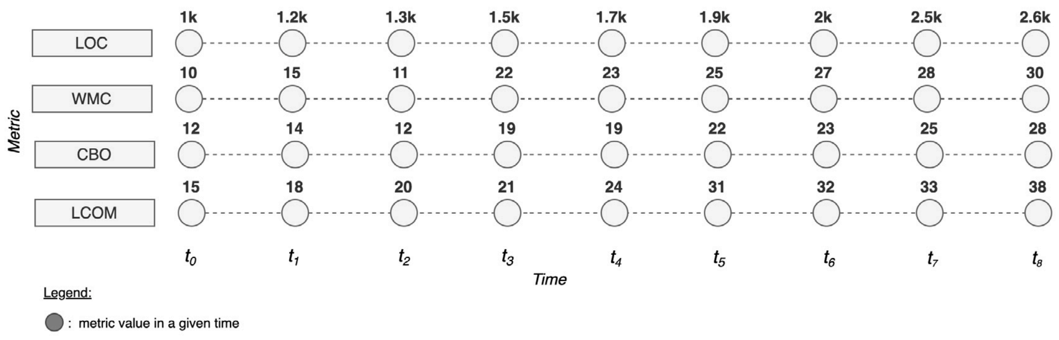

Metric
LOC
WMC
CBO
LCOM
1k 1.2k 1.3k 1.5k 1.7k 1.9k 2k 2.5k 2.6k
10 15 11 22 23 25 27 28 30
12 14 12 19 19 22 23 25 28
15 18 20 21 24 31 32 33 38
t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8
Time
Legend:
● : metric value in a given time

Fig. 1. Motivating scenario.

correctness and low error rate) and efficiency (low effort). In addition, the empirical knowledge generated benefits (2) the research community by providing data-driven insights that enhance understanding and promote future best practices and (3) software developers by providing precise tools for identifying and addressing code smells, thus improving code quality and maintainability.

The article is structured as follows. Section 2 presents some concepts and a motivating scenario. Section 3 describes the related work. Section 4 details the SmellDSL. Section 5 introduces the protocol defined to evaluate the SmellDSL. Section 6 points out the collected results. Section 7 presents additional discussion. Section 8 presents threats to validity. Finally, Section 9 outlines the conclusions and future works.

2. Concepts and motivation

This section introduces some concepts (Section 2.1) and presents a motivating scenario (Section 2.2).

2.1. Code smell and design degradation problems

Code smell patterns. A code smell denotes a suboptimal decision in system implementation, often indicative of underlying design degradation issues [8,9,18]. Code smells can also be seen as an identifiable pattern in the source code that violates coding conventions and challenges design principles or rules, negatively impacting internal quality [19]. Code smells are not bugs but indicate weaknesses in the software's design or implementation, suggesting areas requiring refactoring to enhance maintainability, readability, and extensibility [2]. Since software system documentation is commonly outdated [20], developers use code smells to identify and reason about necessary refactoring operations. When overlooked, design degradation boosts maintenance costs [8]. Certain groups of code smells, known as smell patterns (*e.g.*, *Fat Interface*, *Concern Overload*, and *Scattered Concern*), can also represent code-level problems that may arise shortly after specific design degradation issues. Prioritizing the identification and specification of code smell patterns is necessary for developers to improve software quality and avoid future failures and understanding problems, emphasizing the importance of strategies such as refactoring [6], testing, documentation, and code reviews in mitigating the harmful effects of code smells.

Over the past decade, code smell catalogs and metrics [7,21,22] have been explored to aid developers in identifying software design degradation symptoms. Marinescu [23] defined detection strategies for some code smells using metrics. Lanza et al. [24] also contributed significantly to this area. Bigonha et al. [13] presented detection strategies for code smells such as *Large Class*, *Long Method*, *Data Class*, *Feature Envy*, and *Refused Bequest* based on metrics. For example, the code smell *Long Method* refers to complex methods that undertake multiple responsibilities. Recent works highlight that code smells are found in source code files without the support of any domain-specific language [25–27]. Unfortunately, code smells are often defined and specified informally, complicating the understanding and refactoring efforts. Despite the current efforts in the literature, studies still need to detail how to specify smells correctly, and their characteristics through a clear and detailed explanation, this lack generates the impacts resulting from incorrect coding.

Subjective of code smell patterns. Another vital aspect that demands efforts from academia is developers' divergent views about code smells in a project [17,28]. The subjective nature of how developers define different types of code smells leads to questions about the results of previous studies on code smell detection that were overlooked. To enhance the state-of-the-art techniques for accurate and personalized detection of code smells, developers' correct understanding of code smells is crucial [17]. Finally, the specification of code smells includes a need for more structured and detailed methods defining the tasks users must perform using an application and an adequate interface for this task.

Design degradation problems. Design degradation arises when one or more design decisions compromise these non-functional requirements. Such degradation can manifest through architectural and implementation issues. An architectural problem degrades the system's module decomposition and inter-module communication. An implementation problem, typically represented by code smells, indicates design degradation at the code level [8,29]. Both issues impact software quality by making the software systems as a whole harder to maintain [19].

The software design degrades when symptoms of poor structural decisions (*i.e.*, smells) are detected and introduced by changes [30]. There are studies about developers' perspectives on design degradation, the diversity, and the density of symptoms used (*i.e.*, smells) as characteristics of design degradation [31–33]. Uchôa et al. [34] demonstrated that these bad decisions could arise from problems at various abstraction levels within the software design, whether in a more extensive scope (*i.e.*, system architecture) or a smaller scope (*i.e.*, within a system module design). Thus, developers must interpret the symptoms or code smells, evaluate what constitutes the "disease" and what software design problems are, and efficiently carry out all corrective activities.

Concern metrics. Concern metrics have been defined aiming to capture modularity properties associated with the realization of concerns in software artifacts. Their goal is the identification of specific design flaws or design degeneration caused by poor modularization of concerns. The mapping involves assigning concern to the corresponding design elements that accomplish it [35–37].

Traditional source code metrics are quantitative measures used to assess the complexity, quality, and maintainability of code by analyzing characteristics such as lines of code, cyclomatic complexity, and code duplication. On the other hand, software concern metrics are quantitative measures used to assess and evaluate specific aspects within software systems, typically related to cross-cutting functionalities or features that affect multiple modules or components. These concerns often are related to non-functional requirements such as security, performance, maintainability, or modularity. They may not align directly with a single functional unit but span across different parts of the software architecture. Concern metrics, in turn, aim to provide insights into how well these concerns are addressed within the software, how they influence the complexity of the system, and how they affect its overall quality. Separation of Concerns (SoC) seeks to measure the degree to which a system isolates different concerns into distinct modules, minimizing overlap. Despite being increasingly used in empirical studies, there is a lack of empirical knowledge about the effectiveness of concern metrics to detect code smells [37].

2.2. *Motivating scenario*

The research field of software design has evolved significantly over the past decades. The goal of eliminating source code defects has become synonymous with software quality [38], a perception that can be misleading. A software system could have a flawed or deteriorated design, and merely eliminating defects may not rectify the design itself or justify a complete redesign. Software systems that contain cumbersome and deteriorated hierarchical structures can be inefficient from a developer's perspective, requiring significant refactoring effort. Identifying smelly elements in source code has led researchers toward a more ambitious research agenda for cataloging code smells. However, this focus may have impeded progress on crucial questions regarding construction languages and the support tools that specify composite code smells, especially those that can spread and intertwine between architectural components.

Fig. 1 presents a motivating scenario and illustrates the difficulty of specifying compound code smells and identifying refactoring opportunities as developers understand. Changes in software design are recorded over time. The x-axis represents time (from t_0 to t_8), illustrating nine modifications in software design. The y-axis represents the metrics used to capture such changes in software design — both traditional and concern metrics, *i.e,* LOC (Lines of Code), WMC (Weighted Methods per Class), CBO (Coupling Between Objects), and LCOM (Lack of Cohesion of Methods). Suppose developers need to identify the architectural problem *Misplaced Concern*, composed of the following bad smell patterns: *GodClass*, *Dispersed Coupling*, *Feature Envy*, and *Long Method*. Detection of these smell patterns signals the emergence of *Misplaced Concern*, these indicators of potential problems do not require immediate refactoring corrective action changes to the source code files. The detection of smells can substantially reduce the cost of subsequent activities in the development and maintenance phases [39]. Yet, although such classes "smell", software engineers must manually evaluate their possible negative impact according to the context [40].

The primary challenge is specifying *Misplaced Concern* as a composition of other preexisting smell patterns and determining to what extent the co-occurrence of these smell patterns is tolerated. For instance, the *GodClass* smell pattern can be quantified using metrics [41]. Specifying a *GodClass* poses several challenges for developers: (1) understanding how the metrics are defined, and (2) defining actionable threshold values based on developers' perception, *i.e.*, values that, when exceeded, require immediate refactoring actions. As shown in Fig. 1, developers apply metrics as the source code changes (from t_0 to t_8), with metrics initially measured at t_0 and again at t_8. At t_1, the metrics assume the following values: LOC = 1.2k, WMC = 15, CBO = 14, and LCOM = 18. From t_1 to t_7, all metric values increase significantly: LOC by 108%, WMC by 86%, CBO by 78%, and LCOM by 83%. Metric values can also decrease. For example, at t_2, WMC and CBO metrics dropped by 26% (from 15 to 11) and 14% (from 14 to 12), respectively. According to developers' perception, *GodClasses* may be considered for refactoring when LOC > 2k, WMC > 25, CBO > 19, and LCOM > 31. Specifying *GodClasses* without tool or language support becomes a highly error-prone task.

3. Related work

This section presents how this paper fits within the context of prior studies, identifying approaches, techniques, and tools used for specifying code smells. Related research results were identified in digital repositories such as **Google Scholar**[1] and **Scopus**[2] by applying the search string "(DSL AND (BAD SMELLS OR CODE SMELLS))". In total, five relevant works were selected for further analysis.

3.1. *Analysis of related work*

Bettini et al. (2022) [42]. It used the Eclipse IDE tool to support Edelta 2.0, providing comprehensive DSL static checks during program compilation. The authors' DSL was implemented using Xtext, a popular Eclipse framework for developing programming languages and DSLs. The related work proposes a meta-model that can evolve in code refactoring contexts. Compared to SmellDSL, Edelta is identified as a DSL but fails to address the potential impacts on understanding and addressing incorrect coding of code smells.

Rajkovic et al. (2022) [43]. It introduced NALABS, a desktop application based on .NET standards and packages, developed in C# for the Windows system. It features three layers: (*i*) pre-processing requirements documents stored as Excel spreadsheets, (*ii*) configuration and application of *bad smell* metrics, and (*iii*) displaying results to the final user. This initial experiment proposes future investigation and the definition of other code smell specifications, combining existing specifications with a high-level quality index. This work applies code smells to natural language specifications, setting a specification framework for code smells.

Barriga et al. (2021) [44]. It proposed a PARMOREL extension to aid in code smell detection. This approach selectively removes smells impacting user-defined quality attributes. PARMOREL integrates with a tool allowing modelers to identify and refactor smells. The authors' work fails to analyze the impacts of proposed refactorings quantitatively, unlike SmellDSL, which can specify code smells and their variations according to each software project and user definitions.

Moha et al. (2010) [45]. It conducted a domain analysis on the design concepts of code smells. They evaluated the 2D-DSL language and its structures alongside the 2D-FW language, which can automatically generate detection algorithms. They concluded that the combined use of both languages could aid in specifying code smells at the domain level.

Moha et al. (2009) [40]. It introduced DECOR, a method incorporating all steps required to define detection techniques, and alongside DETEX, a new technique adaptable to the DECOR context for instantiating objects. However, they did not compare DETEX with other techniques and concluded that other existing techniques could generalize DETEX if relevant data for each step were provided.

Additional analysis. The identification of code smells is a very active field of research. Still, its detection and validation process requires a great effort on the part of those involved in the software project and mainly on the part of the developers and how they perceive these anomalies [17,46,47]. Research initiatives currently tend to focus on proposing machine learning techniques [22]. Many classifiers have been explored in the literature, but it has not yet been possible to find an effective model to specify different types of code smells [48]. However, the evaluation of the developers' efforts in specifying code is necessary to evaluate a tool or approach that helps the developer use the proposed techniques and other resources as a still incipient DSL. Thus, the lack of quantitative and qualitative indicators regarding the effort to specify code smells can make it particularly difficult to understand the characteristics of code smells and elaborate on their determined composition and possible variations of smells that help techniques for detecting these anomalies.

[1] Google Scholar: https://scholar.google.com/

[2] Scopus (Elsevier): https://www.elsevier.com/pt-br

Table 1
Related Work Descriptive Comparison.

Criterion	SmellDSL	Bettini et al. (2022) [42]	Rajkovic et al. (2022) [43]	Barriga et al. (2021) [44]	Moha et al. (2010) [45]	Moha et al. (2009) [40]
Tool support	SmellDSL tool	eDelta	Nlabs	Approach	Framework	DECOR/DETEX
Methodology	Controlled experiment	Metamodel approach and applied research, of an experimental nature	applied and experimental research, automatic detection of problems in requirements documents and tests written in natural language	applied and experimental research	applied and descriptive, experimental research, with qualitative and quantitative	applied and descriptive experimental research, with qualitative and quantitative
Contribution	Creation of SmellDSL that improves the specification of code smells	Creation of eDelta, allowing the safe evolution of metamodels	Creation of the Nalabs tool that improves the quality of requirements documents and test specifications	Identification of the balance between removing code smells and maintaining design quality	Establishment of a process to connect domain analysis to smell design, generating systematic identification methods	Formalization of the DECOR method, which offers a systematic approach to identifying code and design smells
Support	Support for integration with Xtext and Eclipse, with industry direction.	Support for metamodel updates	Assistance in identifying and fixing problems in test requirements and specifications	Theoretical and practical support for object-oriented design professionals	Structured support for developers	Practical support for developers and quality engineers
Code Smell Granularity	Specific and variable	Specific to changes that affect the metamodel	Code smells at varying levels	Smells at varying conceptual levels	Smells at various granularities	Smells at various granularities
# Participants	35 participants	Not specified	Not specified	Not specified	Not specified	Not specified
Metrics	Effort, correctness, error rate.	Consistency, efficiency of applying transformations, error rate.	Precision, recall, and effort compared to manual review	Analysis of complexity, cohesion, class coverage, and design maintainability	Cyclomatic complexity, number of classes, and design coherence	Metrics based on the taxonomy of code smells

3.2. Comparative analysis and research opportunity

This section analyzes SmellDSL in comparison with selected studies. We considered the characteristics of SmellDSL as criteria for comparison with the selected related papers, as described in Section 4. The comparison criteria highlight similarities and differences between works. The comparison criteria, inspired by works such as [49,50], are discussed below:

Table 1 summarizes a descriptive comparison of related works on the specification and detection of code smells. The papers are related to the tools SmellDSL, eDelta, and Nalabs, as well as the methods of Moha et al. (2010 and 2009). SmellDSL focuses on developing a DSL to increase developer awareness through the specification of code smells, and eDelta allows the safe evolution of metamodels. Nalabs allows to automatically detect problems in requirements and test documents, in natural language. Barriga et al. (2021) analyze the trade-off between removing smells and maintaining design quality. Moha et al. (2010) suggest a process to connect domain analysis to smell identification and the DECOR/DETEX method formalizes the systematic detection of smells.

To complement the comparison data, Table 1 is shown the descriptive criteria also help to identify relevant factors that may influence the results, enabling a more in-depth analysis. They function as justifications for the decisions made in the development of the research, guiding the choice of methods, selection of samples and variables studied.

We have broadened the discussion to highlight cases in which SmellDSL outperforms traditional methods, such as its flexibility in allowing teams to define custom smells and its user-friendliness in specification tasks. At the same time, we acknowledge its current limitations, particularly in handling more complex detection tasks where traditional metric-based tools may be more efficient. By analyzing both the strengths and weaknesses, we provide a clearer picture of SmellDSL's practical value. Furthermore, we have enhanced the discussion on how these comparisons impact real-world applications, reinforcing SmellDSL's potential role in modern software development practices while identifying areas for future improvement.

4. SmellDSL proposal

This section introduces SmellDSL, a tool-supported domain-specific language designed to assist developers when specifying code smells. SmellDSL enables the developer-sensitive smell specification by providing particular constructs to support the specification of developer detection heuristics. We revisit the motivating scenarios to demonstrate how SmellDSL effectively mitigates the presented problems (Section 4.1). Then, we detail how SmellDSL was designed to be developer-sensitive by design rather than developer-agnostic, highlighting our language design decisions and primary language constructs (Section 4.2). Next, we describe the syntax of SmellDSL (Section 4.3) and which technologies were used to implement a support tool integrated into the Eclipse platform.

4.1. Revisiting the motivating scenario

Although current approaches offer a variety of metrics and heuristics for detecting code smells, recent studies suggest that the commonly triggered and quantified metrics and rules produce an overwhelming number of alerts, leading developers to disregard such signals. Uchôa et al. [34] argue that when concepts are well-defined and understood, such practices can paradoxically burden the development task, suggesting that certain code review practices may increase the risk of project degradation, including protracted discussions and a high rate of developer disagreement. Furthermore, Azeem et al. [51] note that quantifying the *GodClass* smell differs among tools, complicating uniformity across tools, plugins, and IDEs for this identification by various development teams. Despite their accuracy, previous works have identified three significant limitations that may hinder the practical use of smell detectors: (i) the subjectivity of developers regarding detected smells [17], (ii) poor agreement among different detectors [8], and (iii) challenges in establishing effective detection thresholds [13].

Recent empirical studies [8,29] reveal that developers typically recognize design degradation problems when sub-optimal decisions unintentionally compromise internal software quality and when bad

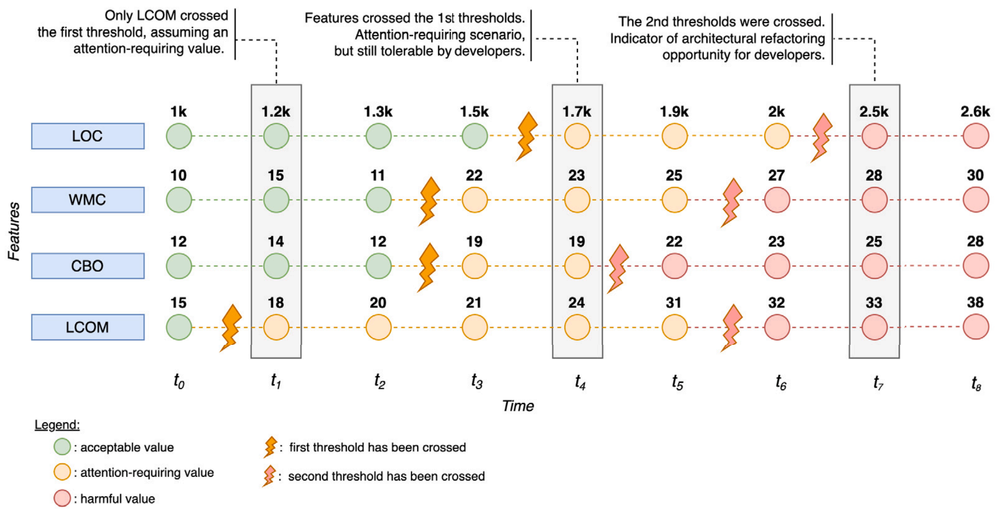

Only LCOM crossed the first threshold, assuming an attention-requiring value.
Features crossed the 1st thresholds. Attention-requiring scenario, but still tolerable by developers.
The 2nd thresholds were crossed. Indicator of architectural refactoring opportunity for developers.
Features
LOC
WMC
CBO
LCOM
1k 1.2k 1.3k 1.5k 1.7k 1.9k 2k 2.5k 2.6k
10 15 11 22 23 25 27 28 30
12 14 12 19 19 22 23 25 28
15 18 20 21 24 31 32 33 38
t0 t1 t2 t3 t4 t5 t6 t7 t8
Time
Legend:
: acceptable value
: attention-requiring value
: harmful value
: first threshold has been crossed
: second threshold has been crossed

Fig. 2. Revisiting the motivating scenario.

smell patterns conspicuously affect many architectural components. This perception varies based on factors such as refactoring effort, types of functionality, and development platforms. We refer to this understanding as a design problem context, which acts as a container for features, usually representing numerical data derived from metrics. Based on their perceptions, developers use this data to specify when severe bad smell patterns emerge, establishing thresholds for such features.

Fig. 2 revisits the motivating example and illustrates the concepts introduced by SmellDSL. These concepts aim to mitigate the issues highlighted earlier (Section 1). In SmellDSL, a *feature* (y-axis) refers to any resource used by developers to characterize code smells, including heuristics and metrics, *e.g.*, LOC (Lines of Code), WMC (Weighted Methods per Class), CBO (Coupling Between Objects), and LCOM (Lack of Cohesion of Methods). The feature values change over time (x-axis) as developers create new functionalities, fix bugs, or implement architectural improvements. For instance, the LOC feature increased from 1k to 2.6k over time (t_{0-8}). In Fig. 2, the first and second thresholds represent *feature-wise thresholds*, *i.e.*, sets of values considered independently for each feature under examination. Developers' perceptions highly influence these threshold sets. At t_1, LCOM crosses the threshold value of 17, indicating an *attention-requiring value* (in yellow). By t_4, all features surpass the first threshold, turning yellow to denote an *attention-requiring scenario*, yet one still tolerable to developers. This means developers prefer to "live" with design problems rather than stop and address their development tasks immediately. By t_7, all features exceed the second threshold, indicating an *refactoring-requiring scenario*. These harmful values of software design metrics can pinpoint severe architectural degradation problems, affecting how the system is decomposed into modules (or architectural components) and how such modules communicate. Moreover, exceeding the second threshold reveals insightful and actionable information on the negative effect of sub-optimal decisions on internal software quality. The SmellDSL provides constructs that help developers specify when such attention- and refactoring-requiring scenarios can appear.

4.2. Language grammar

Fig. 3 shows the syntax diagram of the language elements. The grammar of SmellDSL was defined using BNF notation and incorporated into the Xtext framework.[3] This grammar defines the language constructs explained previously, including `smelltype`, `smell`, `feature`, `symptom`, `treatment`, and `rule`.

A *smelltype* is an abstract code smell that cannot be detected directly, lacks specific detection rules, and serves only to express the concept. Meanwhile, a smell represents a concrete design problem that can be detected with well-defined rules. A smell consists of one or more features, a symptom, and a treatment. A feature is a measurable characteristic used to detect a Smell, with each feature requiring at least one threshold to indicate special attention. For instance, a feature could be represented as a sequence of methods in a class with limits of 50 and 80 line numbers, where 50 lines indicate a point of attention, and 80 suggests immediate refactoring. A smell also encompasses a symptom and treatment, representing unwanted code perceptions and actions to mitigate such perceptions.

Fig. 4 illustrates an implementation using the SmellDSL tool to specify developer-sensitive code smells. In line 1, the *smelltype GodClass* is declared, and in line 2, the class *CompositeGodClass* is an extension of *GodClass* with four features declared in lines 3-6 (LOC, WMC, CBO, and LCOM) with range values (LOW, MEDIUM, and HIGH). In line 7, a *symptom* indicates a class with high complexity and low cohesion. In line 8, a *treatment* informs that the code should be divided into smaller parts. In line 10, the rule *oneattentionrule* is defined, which triggers an "*Attention Possible Code Smell*" alert when one of these four values exceeds the average threshold. This example demonstrates how SmellDSL can identify classes with high complexity and low cohesion, typically considered as code smells.

4.3. Implementation aspects

This work introduces the SmellDSL tool (available in GitHub[4]), a support tool for specifying code smells integrated into the Eclipse IDE[5] platform and implemented using the Xtext framework [52,53]. SmellDSL implementation leveraged the Xtext framework for three main reasons. Firstly, Xtext's effectiveness in facilitating the development of domain-specific languages (DSLs) streamlined the creation process, allowing for precise and tailored language constructs conducive to expressing code smells effectively [53,54]. Secondly, its practicality demonstrated valuable, offering seamless integration with Eclipse IDE and robust language composition and validation support [52,55,56]. Lastly, the widespread adoption of Xtext in various research endeavors

[3] XText framework: https://www.eclipse.org/Xtext/

[4] SmellDSL repository: https://github.com/kleinnerfarias/smelldsl

[5] Eclipse IDE: https://eclipseide.org/

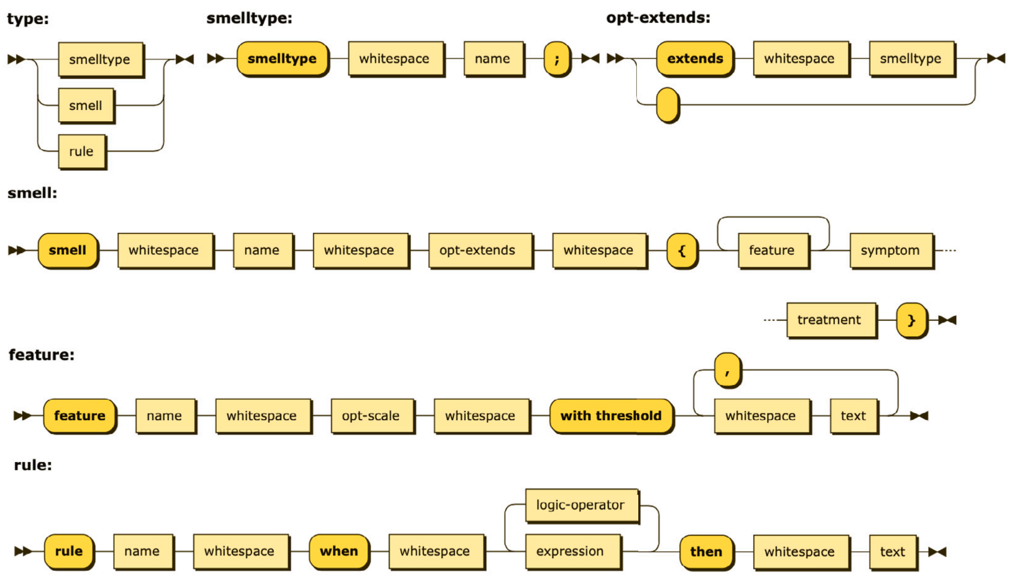
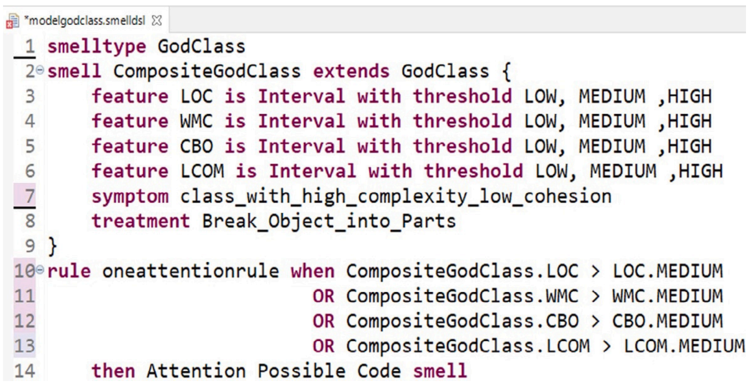


Fig. 3. *Railroad* diagram of the language syntax.



```
smelltype GodClass
smell CompositeGodClass extends GodClass {
    feature LOC is Interval with threshold LOW, MEDIUM ,HIGH
    feature WMC is Interval with threshold LOW, MEDIUM ,HIGH
    feature CBO is Interval with threshold LOW, MEDIUM ,HIGH
    feature LCOM is Interval with threshold LOW, MEDIUM ,HIGH
    symptom class_with_high_complexity_low_cohesion
    treatment Break_Object_into_Parts
}
rule oneattentionrule when CompositeGodClass.LOC > LOC.MEDIUM
                        OR CompositeGodClass.WMC > WMC.MEDIUM
                        OR CompositeGodClass.CBO > CBO.MEDIUM
                        OR CompositeGodClass.LCOM > LCOM.MEDIUM
    then Attention Possible Code smell
```

Fig. 4. Using SmellDSL to specify a developer-sensitive code smell.

validated its reliability and versatility [57–61], instilling confidence in its suitability for our project. The SmellDSL tool provides a suite of resources to facilitate the drafting of code that defines code smells and specifies their occurrence rules as represented in Fig. 5. Key features of the SmellDSL tool include:

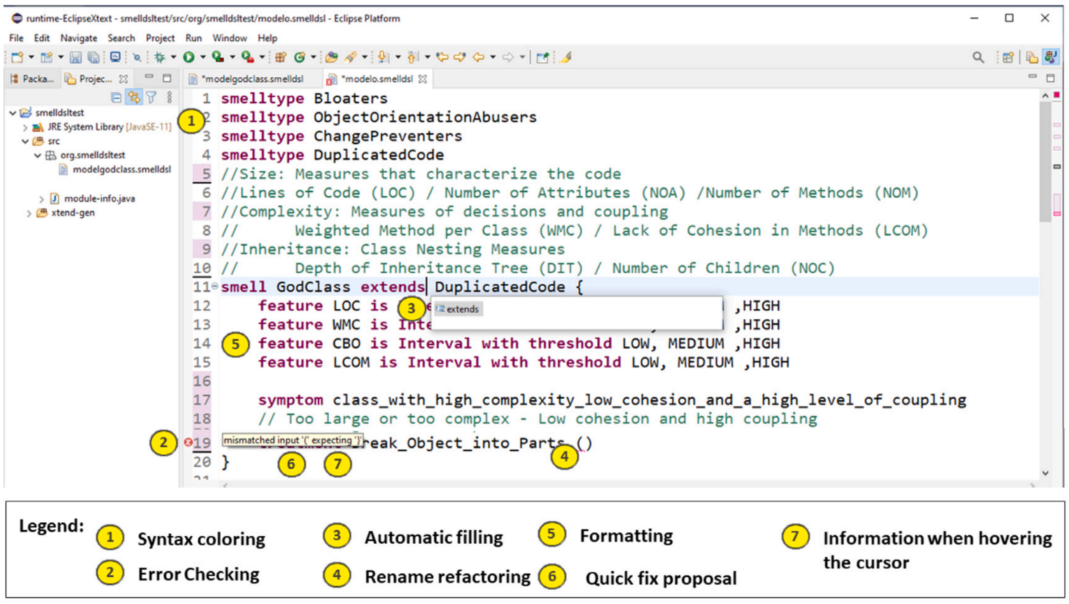
- **Syntax coloring (1):** It is a visual feature in the SmellDSL tool that highlights different elements of code smell script using distinct colors based on their syntactic role, aiding readability and comprehension for developers.
- **Error checking (2):** It is a process performed by the SmellDSL tool to analyze the created code smell scripts for syntax or logic errors, providing feedback to developers about potential issues, such as violations of language syntax or references to undefined elements.
- **Automatic filling (3):** Often referred to as auto-completion, it is a feature in the SmellDSL editor that predicts and suggests SmellDSL elements as developers type, helping to expedite coding tasks and reduce typing errors.
- **Renaming refactoring (4):** It is a programming practice that systematically changes the names of code parts, ensuring consistency and clarity while minimizing the risk of introducing errors. When renamed, it automatically updates all references to a code element.
- **Formatting (5):** It allows the definition and automatic application of code formatting rules, such as line breaks, spaces, and tabs.
- **Quick fix proposal (6):** It is a suggestion provided by the SmellDSL editor to automatically resolve coding errors or improve code quality, offering developers efficient solutions to common issues with minimal manual intervention.
- **Information when hovering the cursor (7):** When hovering the cursor, information concerning contextual details appears. This provides instant insights into SmellDSL code elements, aiding developers in understanding their usage or purpose without needing to navigate elsewhere.

5. Evaluation

This section presents the research protocol used to evaluate the SmellDSL. Our protocol is based on widely adopted empirical guidelines [62] and empirical studies previously published [63–65].

5.1. Objective and research question

This study primarily assesses the effects of SmellDSL on three key aspects: the effort required by developers to specify code smells (*specification effort*), the rate of code smell specifications correctly


runtime-EclipseXtext - smelldsltest/src/org/smelldsltest/modelo.smelldsl - Eclipse Platform

```
smelltype Bloaters
smelltype ObjectOrientationAbusers
smelltype ChangePreventers
smelltype DuplicatedCode
//Size: Measures that characterize the code
//Lines of Code (LOC) / Number of Attributes (NOA) /Number of Methods (NOM)
//Complexity: Measures of decisions and coupling
//      Weighted Method per Class (WMC) / Lack of Cohesion in Methods (LCOM)
//Inheritance: Class Nesting Measures
//      Depth of Inheritance Tree (DIT) / Number of Children (NOC)
smell GodClass extends DuplicatedCode {
    feature LOC is                                   ,HIGH
    feature WMC is Inte                              ,HIGH
    feature CBO is Interval with threshold LOW, MEDIUM ,HIGH
    feature LCOM is Interval with threshold LOW, MEDIUM ,HIGH

    symptom class_with_high_complexity_low_cohesion_and_a_high_level_of_coupling
    // Too large or too complex - Low cohesion and high coupling
                  reak_Object_into_Parts ()
}
```

mismatched input '(' expecting ')'

Fig. 5. The SmellDSL tool integrated into the Eclipse platform.

executed by developers (*correctness rate*), and the error rate identified in the elaborated specifications (*error rate*). We investigate these effects through a controlled experiment with software developers while running code smell specification tasks. To more rigorously define the objective of our study, we use GQM (Goal-Question-Metric) template [62], structured as follows:

Analyze *SmellDSL*
for the purpose of *investigating its effects*
with respect to *effort, correctness, and error rate*
from the perspective of *software developers*
in the context of *specifying code smells.*

GQM played a critical role in shaping our research by providing a structured framework for defining clear objectives and selecting meaningful metrics. This approach ensured that we stayed focused on the key goals of the study and provided a systematic way to evaluate SmellDSL's effectiveness.

In particular, this study aims to evaluate the effects of the SmellDSL on the developer's effort, specification correctness, and error rate while specifying code smells. Thus, we formulate three research questions (RQs) to explore this goal better:

- **RQ1:** How much effort do developers invest in specifying code smells?
- **RQ2:** What is the correctness rate for developers when specifying code smells, calculated as the proportion of correctly specified code smells relative to the total number of code smells specified?
- **RQ3:** What is the error rate of developers when specifying code smells, calculated as the proportion of incorrectly specified code smells to the total number of specified code smells?

5.2. Hypotheses formulation

This section presents a detailed description of the hypotheses formulated based on using SmellDSL. Table 2 shows a general summary of the hypotheses for evaluating the eight experimental tasks performed by each participant. For each research question, we formulate a hypothesis, which is carefully detailed below.

Hypothesis 1. H1: Specification Effort ($\mathbb{SE}$) We postulate that SmellDSL has the potential to significantly reduce the effort required to specify code smells to less than 15 min. This hypothesis is based on the potential of SmellDSL to offer a systematic approach to defining developer-sensitive code smells, potentially simplifying the specification process. The language could empower developers to express complex refactoring scenarios more efficiently, thereby facilitating effective code adjustments and systematic handling of conflicting code parts. However, in the absence of empirical data, we also acknowledge the possibility that using SmellDSL may not effectively decrease the overall specification effort in practice. Developers might invest more effort in specifying code smells using SmellDSL without necessarily increasing the number of accurate specifications. Hence, we present the null hypothesis that SmellDSL does not reduce the effort to specify code smells to less than 15 min, while the alternative hypothesis suggests that it does:

Null Hypothesis 1, H_{1-0}: Developers invest 15 minutes or more to specify code smells using SmellDSL.
H_{1-0}: $\mathbb{SE}(smell)_{SmellDSL} \geq 15$ minutes
Alternative Hypothesis 1, H_{1-1}: Developers invest less than 15 minutes to specify code smells using SmellDSL.
H_{1-1}: $\mathbb{SE}(smell)_{SmellDSL} < 15$ minutes

Hypothesis 2. H2: Correctness Rate ($\mathbb{CR}$). We propose a hypothesis regarding the effect of SmellDSL on the correctness rate for code smell specification. We aim to assess its potential to correctly specify code smells, while also considering the potential challenges in each task when using SmellDSL. The null hypothesis posits that the usage of SmellDSL does not result in a significant increase in the correctness rate for code smell specification, maintaining it at or below the proposed baseline of 50%. This hypothesis acknowledges the uncertainty surrounding the effectiveness of SmellDSL in improving correctness rates, particularly considering potential knowledge gaps and difficulties experienced by developers in adapting to a new language. Additionally, it considers the possibility that the inherent complexity of certain code smells may limit the effectiveness of SmellDSL in ensuring accuracy. However, the alternative hypothesis suggests a more positive outcome,

Table 2
Analyzed hypotheses.

Null Hypothesis	Alternative Hypothesis
H_{1-0}: $\mathbb{SE}$ $(smell)_{SmellDSL}$ $\geq$ 15 min	H_{1-1}: $\mathbb{SE}$ $(smell)_{SmellDSL}$ $<$ 15 min
H_{2-0}: $\mathbb{CR}$ $(smell)_{SmellDSL}$ $\leq$ 50%	H_{2-1}: $\mathbb{CR}$ $(smell)_{SmellDSL}$ $>$ 50%
H_{3-0}: $\mathbb{ER}$ $(smell)_{SmellDSL}$ $\geq$ 50%	H_{3-1}: $\mathbb{ER}$ $(smell)_{SmellDSL}$ $<$ 50%

that SmellDSL significantly increases the correctness rate for code smell specification, resulting in a correctness rate above the proposed baseline of 50%. This hypothesis is supported by the structured approach and systematic guidance provided by SmellDSL, which can potentially enhance developers' understanding and accuracy in specifying code smells. Moreover, using object-oriented concepts within SmellDSL may facilitate developers' comprehension and lead to higher correctness rates. Despite potential challenges, such as knowledge gaps and complexity, this hypothesis proposes that SmellDSL enables developers to produce more accurate code smell specifications due to its detailed syntax and comprehensive language structure. So, the null and alternative hypotheses are formulated below:

Null Hypothesis 2, **H_{2-0}:** Developers produce code smell specifications using SmellDSL with a correctness rate of 50% or less.
H_{2-0}: $\mathbb{CR}(smell)_{SmellDSL} \leq 50\%$
Alternative Hypothesis 2, **H_{2-1}:** Developers produce code smell specifications using SmellDSL with a correctness rate greater than 50%.
H_{2-1}: $\mathbb{CR}(smell)_{SmellDSL} > 50\%$

Hypothesis 3. H3: **Error Rate** ($\mathbb{ER}$). We formulate the hypothesis regarding the effect of SmellDSL on the error rate for code smell specification by considering its potential to reduce errors. This formulation results from a thorough research, considering the potential benefits and challenges associated with its adoption. The null hypothesis states that using SmellDSL does not reduce the error rate for code smell specification, maintaining it at or above the proposed baseline of 50%. This hypothesis reflects the uncertainty surrounding the effectiveness of SmellDSL in minimizing errors, particularly considering the complexity of particular code smells and the learning curve associated with adopting a new language. Conversely, the alternative hypothesis asserts that SmellDSL significantly reduces the error rate for code smell specification, resulting in an error rate below the proposed baseline of 50%. This hypothesis is based on the structured nature of SmellDSL, which provides clear guidelines and a systematic approach for specifying code smells, potentially leading to fewer errors compared to less structured techniques. While acknowledging the potential challenges and initial learning curve, this hypothesis suggests that SmellDSL ultimately enables developers to produce more accurate specifications of code smells, thereby lowering the error rate.

Null Hypothesis 3, **H_{3-0}:** Developers produce code smell specifications using SmellDSL with an *error rate* greater than or equal to 50%.
H_{3-0}: $\mathbb{ER}(smell)_{SmellDSL} \geq 50\%$
Alternative Hypothesis 3, **H_{3-1}:** Developers produce code smell specifications using SmellDSL with an error rate of less than 50%.
H_{3-1}: $\mathbb{ER}(smell)_{SmellDSL} < 50\%$

By testing the three hypotheses formulated (Table 2), we produce empirical knowledge about the effects of SmellDSL on the effort to specify code smells, the rate of errors found in the specifications, and the rate of correctly specified code smells. With this knowledge, researchers and software developers can decide whether to use it (or not) based on initial empirical evidence. These hypotheses were defined based on current literature that addresses performance standards to identify code smells in similar tasks [66,67].

5.3. Study variables

Specification Effort ($\mathbb{SE}$): The effort invested (minutes) by developers in specifying code smells is the dependent variable of the first hypothesis. This variable is an essential indicator related to the time developers spend using SmellDSL to specify the features of undesirable code structures within a software system. By quantifying the effort required for code smell specification, the study aims to evaluate the effectiveness and efficiency of SmellDSL as a mechanism to facilitate code smell representation at the developer level. This measure provides not just insights but practical implications for the utility and usability of SmellDSL from a developer-centric perspective, informing future enhancements and refinements of the tool to better meet the needs of software development practitioners.

Correctness Rate ($\mathbb{CR}$): For each code smell specification request, participants are required to specify the code smell using SmellDSL. The aim is to match the developer-sensitive requirements, which are specifications that align with developers' perceptions of code smells. The correctness of these specifications is determined by their complete adherence to the predefined requirements (Eq. (1)). The dependent variable of the second hypothesis is the correctness rate (Eq. (2)), ranging from 0 (indicating low correctness) to 1 (indicating high correctness), which is calculated as the average of correct specifications across the experimental tasks (Table 3). This rate measures the specifications' correctness, ensuring a comprehensive evaluation process. Eq. (1) defines how *correctness* is computed in each experimental task. It assigns 1 to an experimental task performed *correctly* and 0 when *incorrectly*. Eq. (2) calculates the *correctness rate* of the experimental tasks performed by the participants. If 30 out of 35 participants correctly answered the first experimental task, the correctness rate would be 0.85 (or 85%), representing the proportion of correct responses relative to the total number of participants. A high correctness rate indicates ease in specifying bad smells, while a low correctness rate indicates difficulty in specifying bad smells.

$$Correctness(task) = \begin{cases} 1,\ if\ all\ steps\ are\ correct;\ error < 0 \\ 0,\ if\ there\ is\ an\ error\ in\ any\ step \\ \quad performed;\ error > 0 \end{cases} \tag{1}$$

$$CorrectnessRate(t)overall = \frac{\sum_{k=0}^{j-1}(Correctness(t))}{j} \tag{2}$$

$Legend$:j: number of participants, t: task

Legend: j: number of participants, t: task

Error Rate ($\mathbb{ER}$): The third hypothesis focuses on the error rate associated with specifying code smells. This variable, ranging from 0 to 1, quantifies the proportion of errors within a code smell specification. A value of 0 indicates a low error rate, while 1 indicates a high error rate. In Fig. 4, a correct specification yields an error rate of zero, while any incorrect step performed would increase the error rate. This measure is particularly relevant for providing insights into the effectiveness of the SmellDSL when participants make mistakes. An error analysis must consider the sum of all errors concerning the number of steps performed per task described in Section 5.5. Despite variations in error rates between participants, the average may indicate consistent overall performance across the 280 evaluation scenarios. Each metric must be considered separately to provide a complete understanding of the participants' performance. Eq. (3) defines the errors of a task as the ratio between the number of correct steps and the total number of steps.

Eq. (4) calculates the overall error rate by accumulating errors from multiple tasks divided by the total number of tasks.

$$Error(task) = \frac{Number\ of\ correct\ steps}{Total\ number\ of\ steps} \tag{3}$$

$$ErrorRate(t)overall = \frac{\sum_{k=0}^{j-1}(Error(t))}{j} \tag{4}$$

Legend :j: number of steps t: task

5.4. Experimental process

The experimental process, a crucial component of this study, was meticulously designed to adhere to established empirical study guidelines [62,64,68,69]. It comprised three distinct phases, as depicted in Fig. 6. These phases were carefully crafted to minimize biases throughout the experiment, ensuring the reliability and validity of our findings. To perform the activities in each step, we applied a mixed-method approach to analyze whether and how developers can effectively find code smells based on [70]. To perform the experimental tasks, participants received the following material:

- A brief textual description of how to specify code smells with SmellDSL;
- The system's source code and an Integrated Development Environment to navigate it (Eclipse);
- Document with code smell stereotypes to be specified;
- A questionnaire containing participants' understanding questions and information;
- A post-experiment survey questionnaire;

According to the design of the experiment carried out, each participant was guided and trained to perform 8 experimental code smell specification tasks, each participant performed the tasks in approximately 2.5 h. The experimental process adopted is explained as follows:

- **Phase 1: Training and familiarity.** Participants underwent a comprehensive training program. This training encompassed a deep understanding of the SmellDSL grammar and technical aspects of specifying code smells. We conducted a *training and pilot study* to ensure participants' thorough readiness for subsequent tasks. Ensuring all participants were fully prepared and confident to face the subsequent tasks. Through this training process, we establish a solid foundation so that participants can apply their knowledge effectively while carrying out experimental tasks.
  Activity 1: Apply the questionnaire. Describe the characteristics of the subjects. The questionnaire used to characterize the subject consisted of questions that described the characteristics of each participant, including academic background, professional experience in programming, and length of experience in software modeling. Completing the questionnaire was an activity that required approximately 15 min per participant based on [70]. This activity was essential to obtain a comprehensive profile of those involved in the study, ensuring that relevant variables were adequately considered in the subsequent analysis.
  Activity 2: Training session. After defining the order in which each step should be carried out, the next step is to train the participants. The main objective of the training sessions is to place participants in the same context necessary to understand and adequately execute the experimental tasks. Therefore, they are trained in basic concepts and terminology *description of bad smells*. Each participant received this training only once before the experiment's first phase. The **training** includes 15 min presentations covering the following topics: software design, code smells, and design problems. Training sessions last approximately 15 min, and participants are encouraged to ask questions and voice any concerns, thus promoting an interactive learning environment. This approach aims to not only enrich participants' understanding but also effectively prepare them to contribute to the experiment's success, as highlighted on [70].
- **Phase 2: Practical Application of SmellDSL.** This phase was the heart of our study, focusing on the real-world application of SmellDSL to specify code smells according to predefined requests. Each experimental task (Table 3) was accompanied by specific requirements, with controlled variables meticulously measured throughout the process. Video and audio recordings were employed to document task execution for subsequent analysis.
  Activity 3: Introduction to the system. We asked participants to read the document containing the **code smells specification request**. They had 15 min to read the description and implement each task. This time was essential for participants to familiarize themselves with the content, allowing them to begin the design with an adequate level of understanding of SmellDSL. This step aims to ensure that they not only understand the concepts but also feel confident in applying the knowledge acquired in the activities before developing the smell code specifications.
- **Phase 3: Qualitative and Quantitative Evaluation.** The collected data underwent an analysis using statistical methods. Furthermore, a multifaceted approach was adopted, including semi-structured interviews [62], enriching the qualitative dataset collected, and the Technology Acceptance Model (TAM) questionnaire [71], to measure participants' acceptance and usability of SmellDSL.
  Activity 4: In this activity, each participant received a carefully designed feedback form. This form contains a **list of questions** that allows the participant to share their perceptions and opinions about the design of code smells. The questions address various aspects of the process, such as the clarity of the instructions, the relevance of the concepts presented, and the ease of application in practice. This approach targets many important insights that can be used to enhance future training sessions and to refine the code smell design methodology, ensuring that participants' needs and experiences are considered [70].

5.5. Experimental tasks

Table 3 outlines the eight experimental tasks employed in our study. Each participant undertook all tasks, resulting in 280 evaluation scenarios. In Task 1 (T01), each participant must specify a SmellType (*i.e., Bloater*) and a smell (*i.e., LargeClass*). To do this, they must perform one experimental step, which consists of specifying one feature using the SmellDSL. T01 does not aim to maintain bad smells. In Task 8 (T08), each participant must perform a maintenance task involving two *SmellTypes* (*i.e., Bloater* and *Dispensables*) and three *Smells* (*i.e., Large Classes*, *Long Parameter List* and *Feature Envy*). For this, each participant must perform four steps for specifying *feature*, *symptom*, *treatment*, and *rule*. For a task to be considered correctly performed, a participant must execute all experimental steps accurately. If a participant performs incorrectly an experimental step, the task is considered incorrect, and the error is calculated based on the number of steps involved in the task.

The bad smells, presented in Table 3, were chosen because they are widely explored in the literature [2,21,72,73]. For example, Lacerda et al. [21] indicate the most frequent code smells with their detection approaches, detection tools, and suggested refactoring. These smells were classified based on human perception, metrics, and identification strategies/rules. The selected smells represent common challenges in software development and cover several architectural issues. Moreover, studies on identifying code smells show that the effort invested by developers can vary greatly, with an average of between 10 and 45 min for experimental tasks [70,74–78].

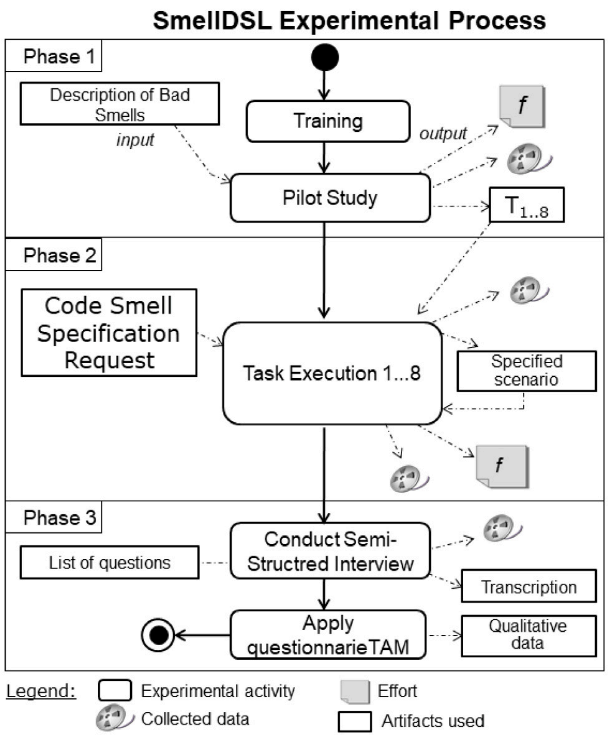


Fig. 6. The adopted experimental process.

Table 3
Experimental tasks.

Task	SmellType			Smell					Experimental Step				Aim
	Bloaters	Dispensables	Couplers	LargeClass	LongParameterList	DuplicateCode	LongMethod	FeatureEnvy	Features	Symptom	Treatment	Rule	Maintenance
T01	●	○	○	●	○	○	○	○	●	○	○	○	○
T02	●	○	○	○	●	○	●	○	●	○	○	○	○
T03	○	●	○	○	○	●	○	○	●	●	●	●	○
T04	○	●	○	○	○	○	○	●	●	●	●	●	○
T05	○	○	●	○	○	○	○	●	●	●	●	●	○
T06	○	○	●	○	○	○	○	●	●	●	●	●	●
T07	●	●	○	●	●	○	○	●	●	●	●	●	●
T08	●	●	○	●	●	○	○	●	●	●	●	●	●

(●) *Support* (○) *Does not support.*

5.6. *Context and selection of participants*

The demographic characteristics of participants significantly impact the generalizability of research findings in software engineering [79–81]. While existing guidelines emphasize the importance of recruiting the right participants, there is limited guidance on achieving this effectively [80]. To enhance the generalizability of our results, we followed known guidelines [80,81] to support our context definition and participant selection, involving defining desired population characteristics and identifying sampling sources to minimize the distance, which refers to the degree of dissimilarity, between the sample and the population.

In this sense, we invited 35 participants by convenience, including professionals and students (with industry experience) from Brazilian software development firms, who were invited via email. Students were recruited in the controlled experiment due to their foundational knowledge, which ensures a consistent baseline of skills and availability, facilitating a more controlled and homogeneous sample for evaluating the effectiveness of SmellDSL. Previous studies showed similar performances between students and professionals [82,83]. Convenience samples, often used in research, may not accurately represent the population under study [84]. However, we restrict our findings to the population considered. This diverse participant pool encompassed individuals holding MSc and bachelor's degrees or equivalents and substantial proficiency in software maintenance tasks and object-oriented programming. The selection process aimed to ensure a heterogeneous mix of profiles and expertise levels among participants (but keeping a distance), enriching the study's findings with varied perspectives and insights. The experiment adhered to stringent quality standards akin to practical laboratory exercises, with each participant undergoing comprehensive training on the proposed SmellDSL and the experimental procedures, ensuring the reliability and validity of the execution of our experimental tasks. This approach aimed to mitigate potential biases and ensure consistency in participant understanding and engagement throughout the evaluation process.

A more detailed discussion of their experience with code smells and software refactoring is essential for accurately interpreting our results. By addressing this aspect, we aim to provide more precise insights into the results and enhance the understanding of how developers' expertise may affect their interaction with the tool.

5.7. *Analysis procedure*

Quantitative analysis. We performed descriptive statistics to analyze its normal distribution and statistical inference to test the hypotheses. First, we conducted a descriptive analysis to assess the distribution, dispersion, and trends, such as means and medians of the data collected. Then, we run statistical analyses to assess all hypotheses, with a significance level set at $\alpha = 0.05$. The single sample Student's t-test was employed to evaluate the hypotheses H1, H2, and H3. The single-sample Student's t-test was chosen for its appropriateness in analyzing the mean difference between a sample and hypothesized population mean, aligning with the within-subjects experimental design [62] is typically employed and the study's objective to assess the efficacy of SmellDSL against a predetermined benchmark. We used the WINKS Statistics tool[6] to run descriptive statistics and statistical inference testing. Remember that participants performed experimental tasks, and we compared their collected data against predetermined.

Qualitative analysis. Various data sources contributed to the qualitative data collection, including questionnaires, audio/video recordings, transcriptions, think-aloud comments, and interviews. This multifaceted approach enabled the acquisition of complementary evidence to elucidate the quantitative findings. Lastly, we assembled a chain of evidence by systematically aligning the quantitative and qualitative data, facilitating the derivation of tangible conclusions.

To assess participants' acceptance and usability of SmellDSL, we used the Technology Acceptance Model (TAM) questionnaire (Table 4) [71]. This model evaluates three key criteria: *perceived ease of use*, which measures how much users believe the technology reduces their effort; *perceived usefulness*, which gauges how much it can enhance their development activities; and *behavior intention*, which seeks to capture the intention of using the proposed technology. Each statement in the questionnaire offered response options on a standard 5-point Likert scale, allowing participants to express their views as follows: Completely Disagree, Partially Disagree, Neutral, Partially Agree, and Completely Agree. The detailed statements included in the questionnaire can be found in Table 4.

5.8. *Replication package*

All our artifacts are publicly available in the *replication package* [85]. The package includes several key elements: the datasets used in the study, detailed setup instructions for replicating the experimental environment, and all necessary code and scripts to run the experiments. We

[6] Winks Statistics: https://www.alanelliott.com/TEXASOFT/

Table 4
Collected data related to TAM questionnaire.

ID	Description
1	*Perceived ease of use* I found SmellDSL easy to use I found SmellDSL easy to learn I found SmellDSL easy to master
2	*Perceived usefulness* SmellDSL would make software maintenance easier SmellDSL would help with productivity SmellDSL would reduce code anomaly identification time
3	*Behavior intention* I would use SmellDSL as a support tool for automatic software maintenance

have also provided thorough documentation explaining the procedures followed during the study. The package contains not only the source code but also the data collected during the experiments, enabling comprehensive verification of the results. By making these resources available, we aim to facilitate the reproducibility of our research, allowing the research community to analyze, replicate, and build upon our findings. We hope this package can encourage further research and lead to new questions, methodological improvements, and deeper insights into the concepts explored in our study.

6. Results

This section presents the collected results after running a controlled experiment to evaluate the SmellDSL. First, we analyze the participant profile (Section 6.1) and discuss the obtained results, considering the formulated research questions (Section 6.2, Section 6.3 and Section 6.4). Next, we explore the effects of participants' experience on our results (Section 6.5) and discuss the results of the TAM questionnaire (Section 6.6).

6.1. Participant profile

Screening. We asked each participant to confirm their experience with software development and modeling so that they could participate in the study. We used convenience sampling to select participants [62]. All participants, including industry professionals, graduate students, and undergraduate students, were notified via email. Our study recruited 35 participants, professionals from Brazilian companies and students with professional experience. Participants received the same training level regarding the proposed DSL's main characteristics for specifying code smells. The professionals had master's and bachelor's degrees (or the equivalent) and had excellent knowledge about software modeling and programming in practice. Undergraduate students participated in our evaluation, which included subjects from different backgrounds and levels of specialization. Students had a solid background in software modeling and a practical knowledge of software development gained through professional experience on real-world business projects. Participants received training on the proposed DSL through practical scenarios of specifying code smells with the developed tool. This way, we ensure that each participant understands how to use the proposed SmellDSL.

Participants' profiles. Table 5 describes the participants'profiles, reporting their main characteristics, including age, education, level of education, academic learning time, overall professional experience with software modeling and development, current position, and current position in years. In total, 35 participants responded to the formulated questionnaire.

Education and Undergraduate Course. The majority (77.1%) had a complete graduate. 5.7% had a Master' s degree in the field of computing. Regardless of their level of education, all participants were professionals with development experience acquired from real-world projects with fast-changing enterprise applications. All participants took an undergraduate course in computing, including Computer Science (25.7%), System Analysis (14.3%), Computer Engineering (5.7%), Information Systems (31.4%), and Others (22.9%). This data highlights the participant's robust academic foundation, complementing their practical experience. The diversity of participant profiles ensures a comprehensive and inclusive study.

Academic Learning Time and Age. We also considered academic learning time and age as key factors in constructing our sample. The first refers to how long our participants were actively engaged in learning relevant academic content. Over 40.0% indicated having five years (or more) of active learning in undergraduate and graduate courses. 45.7% of the participants were between 18 and 25 years old, while the others were between 26 and 45. That is, the age of the participants is distributed rather than concentrated. Therefore, the recruited participants had experiences and life experiences in a broad spectrum.

Overall Experience. Over 37.1% indicated having five years (or more) of professional experience, of which 11.4% had more than eight years of practical experience in realistic software development scenarios in Brazilian companies. The data account for a lower concentration (48.5%) in the experience level of up to 4 years. The sample does not include only experienced participants—which could represent a biased sample. Instead, it contemplates people with different levels of experience, allowing perception of the practical usefulness of the proposed approach to be gathered from a broader spectrum of experience levels.

Current Position. Regarding the current position of the participants, the majority (82.8%) held positions directly related to software development, including programmers (65.7%), systems analysts (11.4%), and software architects (5.7%). The other participants held positions related to software development rather than directly.

Suitable Profile for SmellDSL Evaluation. An ever-present concern was whether or not the participant profiles were suitable for the proposed assessment. To mitigate this threat, we carefully chose participants with practical and academic knowledge, conducting a thorough screening process. In this sense, the participant profile is suitable for our study for two reasons. Firstly, the participants have a solid academic background in undergraduate and postgraduate courses in computer science. This theoretical basis mitigates the risk that a lack of knowledge motivates the unperceived usefulness of carrying out tasks to specify code smells. Secondly, the participants had professional experience in companies and predominantly held different software development roles. It avoids biased responses due to specific positions or those motivated by a lack of practical experience. It enhances the capture of experiences in different positions, levels of experience, and age. In this sense, our sample is adequate to evaluate the proposed DSL, considering the level of acceptance and usefulness of the SmellDSL DSL in more realistic scenarios.

6.2. RQ1: SmellDSL and specification effort

Descriptive Statistics. This section presents an analysis of the gathered data concerning the effect of SmellDSL on developers' efforts. Fig. 7 shows the effort invested to run eight experimental tasks. The x-axis represents the experimental tasks, while the y-axis is the specification effort.

Table 6 presents the descriptive statistics highlighting the data distribution, capturing its central tendencies and variability. Note that these statistics were derived from 280 evaluation scenarios. The principal observation indicates that *participants invested little effort in specifying code smells* when employing SmellDSL compared to the hypothesized effort of 15 min. Specifically, participants showed an average effort of approximately 5 min (Mean = 5.03), with a standard deviation of 4.50. This finding suggests that participants could complete the experimental tasks within the foreseen time frame of 15 min. The next step aims to

Table 5
Profile of participants.

Characteristics	Description	%
Age	18 – 25 years	45.7%
	26 – 35 years	31.4%
	36 – 45 years	20%
	> 45 years	2.9%
Academic Education	Computer Engineering	5.7%
	Information System	31.4%
	Systems Analysis	14.3%
	Computer Science	25.7%
	Others	22.9%
Level of Education	Technician	17.1%
	Graduation	42.9%
	Master	5.7%
	Others	34.3%
Current Position	Programmer	65.7%
	Analyst	11.4%
	Manager	11.4%
	Others	11.5%
Time in Office	< 2 years	20%
	2 – 4 years	20%
	5 – 6 years	25.7%
	> 7 years	34.3%
Experience in Software Modeling	< 2 years	17.1%
	2 – 4 years	17.1%
	5 – 6 years	28.6%
	7 – 8 years	20%
	> 8 years	17.1%
Experience in Software Development	< 2 years	17.1%
	2 – 4 years	17.1%
	5 – 6 years	28.6%
	7 – 8 years	20%
	> 8 years	17.1%

Table 6
The results of descriptive statistics and statistical inference for RQ1.

Task	Descriptive Statistics								Statistical Inference		
	N	Min	25th	Median	75th	Max	Mean	SD	t	D.F.	p-value
All	280	1	2	4	6	32	5.03	4.5	−36.89	279	0.001
1	35	1	2	3	5	14	4.05	2.75	−23.51	34	0.001
2	35	2	3	4	6	11	4.42	2.07	−30.12	34	0.001
3	35	1	4	5	10	24	7.31	5.47	−8.31	34	0.001
4	35	1	2	4	6	30	5.42	5.57	−10.15	34	0.001
5	35	1	3	5	10	32	7.11	5.96	−7.82	34	0.001
6	35	1	2	3	5	12	4.11	3.27	−19.64	34	0.001
7	35	1	1	2	5	18	3.62	3.9	−17.23	34	0.001
8	35	1	2	3	6	23	4.4	4.19	−14.95	34	0.001

Number of tests (N), Standard deviation (SD), 25th quartile (25th), 75th quartile (75th), Arithmetic mean (Mean),
Maximum value (Max), Standard Deviation (SD), t-test value (t), Degrees of freedom (D.F), Statistical significance (p-value).

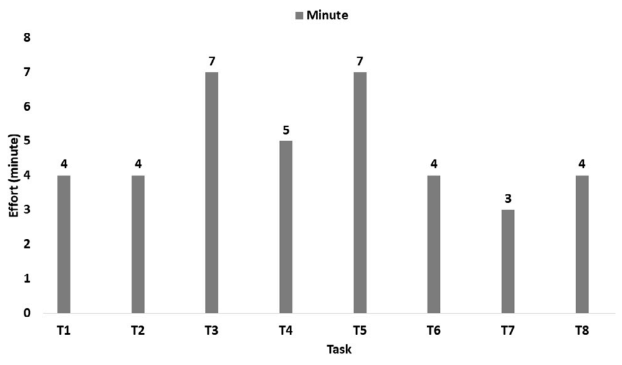


Fig. 7. Effort (minute) invested in each experimental task (RQ1).

examine whether this evidence is statistically significant to reject the first null hypothesis.

Hypothesis testing. Statistical tests were conducted to evaluate the significance of differences in specification effort measures. Our hypothesis posited that SmellDSL required less effort than the hypothesized population mean of 15 min. As previously stated, a significance level of 0.05 (*i.e.*, p-value ≤ 0.05) was chosen to denote true significance. Given the absence of deviations from normality according to the Shapiro–Wilk test, the single-sample Student's t-test was employed. The results indicated $t = -36.89$, with 7 degrees of freedom and p-value < 0.001, showing the difference between the sample mean and the hypothetical 15 min value to be statistically significant.

Summary of RQ1: Specification Effort. A single sample Student's t-test was conducted to evaluate whether the mean effort required to specify code smells using SmellDSL differed significantly from the hypothesized population mean of 15 minutes. Results indicated that the mean effort required to specify code smells using SmellDSL was significantly lower than the hypothesized value of 15 minutes (M = 5.03, SD = 4.5, t = -36.89, df = 279, p < 0.001). The effect size was 2.2, indicating a large effect. These findings suggest that participants invested significantly less effort in specifying code smells using SmellDSL than the hypothesized value. Thus, the null hypothesis was rejected, which stated that utilizing SmellDSL for code smell specification results in an effort of 15 minutes or less per task. Our initial results supported the alternative hypothesis, indicating that SmellDSL can reduce the effort needed to specify code smells.

6.3. RQ2: SmellDSL and correctness rate

Descriptive Statistics. This section analyzes the collected data concerning the impact of SmellDSL on the specification correctness rate. Fig. 8 shows the correctness of the specifications generated using the SmellDSL throughout the eight experimental tasks. Note that the correctness rate is shown as a percentage. The x-axis consists of the experimental tasks, while the y-axis represents the proportions of incorrect and correct specifications achieved by the number of specifications realized in each task. Thus, the histogram shows how the correctly specified code smells happened throughout the experimental tasks. The main outstanding feature is a distribution pattern of the proportions of correctly written specifications in the tasks. We observed a correct specification rate greater than 80% across all experimental tasks. For example, in Task 1, the participants produced a correct specification rate of 88%. We also observed this high correctness rate in the other tasks, which is not a particularity of Task 1. Furthermore, the difficulty level of the tasks did not influence the correctness rate as we observed high rates in easy tasks and more complex tasks involving the creation or maintenance of elaborated SmellDSL code. One interesting insight was that the participants invested little effort in creating and maintaining the SmellDSL code, suggesting that the tool is user-friendly and requires minimal training for a high correctness rate.

Hypothesis Testing. RQ2 evaluates whether using SmellDSL can produce a correct specification rate equal to (or higher than) a conjectured value of 50%. We test H2 to investigate RQ2 by applying the chi-square goodness-of-fit test (χ^2). The second null hypothesis, H_{2-0}, states that no significant difference exists between the proportions of correctly specified code smells produced using the SmellDSL and the conjectured value of 50%. Table 7 shows the collected values. The crucial finding is that, in all tasks, the p-value was low ($p = 0.001$), allowing us to reject H_{2-0}. This suggests that there is a significant difference between the proportions of correctly specified code smells produced using the SmellDSL and the conjectured value of 50%.

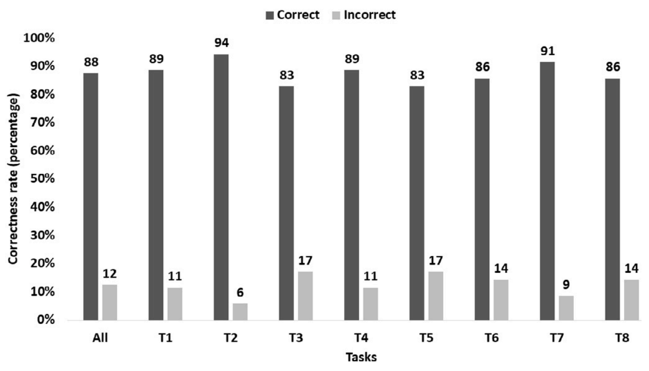


Fig. 8. The correctness rate (RQ2) as a percentage.

Table 7
Chi-square Goodness of Fit test for the correctness rate.

Task	Goodness-of-Fit Statistics			
	N	χ^2	D.F.	p-value
All	280	154.52	1	0.001
1	35	19.33	1	0.001
2	35	25.74	1	0.001
3	35	13.84	1	0.001
4	35	19.33	1	0.001
5	35	13.84	1	0.001
6	35	16.47	1	0.001
7	35	22.42	1	0.001
8	35	16.47	1	0.001

Number of tests (N), Statistical hypothesis, Chi-Square (χ^2), Degrees of freedom (D.F), Statistical significance (p-value).

Summary RQ2: Correctness Rate. A chi-square goodness-of-fit test was conducted to determine whether the average correctness rate for specifying code smells using SmellDSL was significantly different from the hypothesized baseline of 50%. The results indicated that the correctness rate for code smell specifications using SmellDSL was significantly higher than 50% (χ^2 = 154.52, p-value < 0.001). This finding suggests that participants were able to specify code smells with a much higher accuracy than the conjectured value of 50%. Therefore, we rejected the null hypothesis, which stated that the correctness rate would be 50% or lower, and supported the alternative hypothesis that SmellDSL can assist participants in properly specifying code smells.

6.4. RQ3: SmellDSL and error rate

Descriptive Statistics. In this section, we examine the statistical data collected on errors in the specification of code smells. We evaluated the 280 evaluation scenarios performed by participants to analyze the error rate associated with their specifications. Fig. 9 illustrates the distribution of error rates across eight experimental Tasks (T1 to T8) along the X-axis, with values ranging from 0% to 33% on the Y-axis. T1 and T2 show the highest error rates, both reaching 33%, indicating that participants find these tasks more difficult. A significant decrease in the error rate is observed in T3, where the value drops to 17%, demonstrating enhanced task performance. T4, T5, and T6 maintained a relatively moderate error rate of 20% to 25%, indicating stabilization in participant errors. However, T7 worsens performance slightly, with the error rate increasing to 27%. T8 concludes the series with an error rate of 23%, reflecting a slight variation from T7. Overall, the trend indicated by the dashed line suggests a general decline in error rates across the tasks, albeit with some fluctuations, underscoring variations in task complexity or participant performance throughout the experimental sequence.

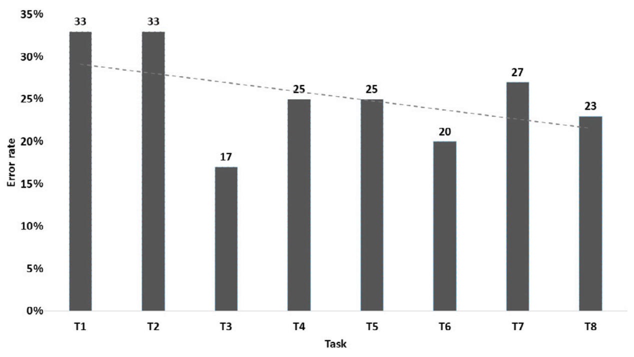


Fig. 9. Error rate (RQ3).

Table 8
Comparative incorrect tasks analysis.

Task	Statistical Inference			
	N	t	D.F	p-value
All	280	−94.09	279	0.001
1	35	−25.67	34	0.001
2	35	−36.63	34	0.001
3	35	−39.96	34	0.001
4	35	−29.46	34	0.001
5	35	−26.57	34	0.001
6	35	−36.96	34	0.001
7	35	−73.93	34	0.001
8	35	−31.17	34	0.001

Number of tests (N), t-test value (t), Degrees of freedom (D.F), Statistical significance (p-value).

Hypothesis Testing. Statistical tests were conducted to assess the significance of the error rate (ER) measures. The hypothesis posited that the error rate for specifying smells using SmellDSL should be less than 50%. A significance level of 0.05 (*i.e.,* p-value ≤ 0.05) was considered to demonstrate validity. Table 8 demonstrates the detailed Incorrect Task Benchmark Analysis data. These findings suggest that participants can better specify code smells using SmellDSL than the hypothesized value. Therefore, the null hypothesis was rejected, which stated that using SmellDSL to specify code smell errors results in a rate greater than 50%. The alternative hypothesis was supported, which proposed that SmellDSL assists in code smell specification.

Summary RQ3: Error Rate. A single sample Student's t-test was conducted to evaluate whether the error rate required to specify code smells using SmellDSL differed significantly from the hypothesized population mean of 50% errors. The results showed an average error rate of 0 (zero) and standard deviation of 0.08, with a t = -94.09 and a p-value of 0.001, decreasing statistical significance 0.05 (i.e., p-value ≤ 0.05). This suggests that the observed error rate is significantly lower than the hypothesized 50% and a 95% confidence interval.

6.5. Analysis of the effects of experience

Our participants' experience levels can influence the SmellDSL's effectiveness. We discuss their impact on the controlled variables. We conducted this analysis based on previous studies [86]. Considering the experience level, students were classified as *Novices*, while those employed in the industry were classified as *Professionals*. Our classification used the questions in Table 9.

Fig. 10 compares *30 novices and 5 highly qualified professionals*. We can observe that *novices invested less effort than professionals in almost all experimental tasks*, except T8. We noted that the average effort is higher

Table 9
Questions of the post-experiment qualitative questionnaire.

ID	Question
Q1	How old are you?
Q2	What is your academic background?
Q3	How long did you study (or have you studied) at university?
Q4	If you are employed, what is your position (role)?
Q5	How long have you played this position (role)?
Q6	How long have you held this role/position?
Q7	What is your practical experience in software modeling?
Q8	What is your practical experience in software development?

A: < 2 years, B: ≥ 2 and < 4 years, C: ≥ 4 years and < 6 years, D: ≥ 6 years and < 8 years, E: ≥ 8 years.

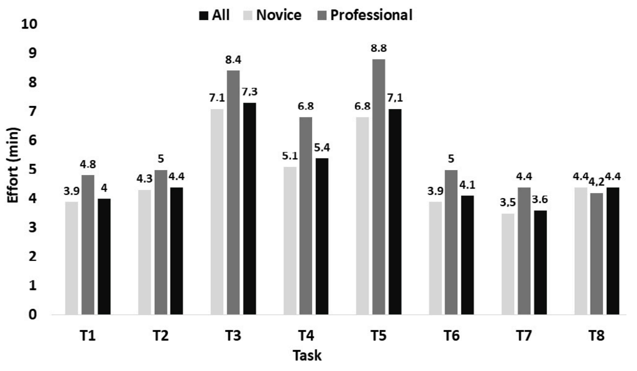


Fig. 10. Effort (minute) for experience and ability.

for professionals compared to novices across almost all tasks, with T3 and T5 requiring the highest effort from professionals. The boxplots in Fig. 11 graphically represent our results considering experience levels. Additionally, Table 10 shows the complete descriptive statistics of the dataset obtained among this selected group.

The box plot visualization compares effort, correctness rate, and error rate between novice and professional participants. In terms of effort, professionals tend to spend more time completing tasks compared to novices, with a higher median and wider interquartile range (IQR), indicating greater variability in effort. The correctness rate is slightly higher for novices (mean = 98.29) than professionals (mean = 97.00), though both groups exhibit a high overall correctness rate. However, the error rate is significantly lower for professionals (mean = 0.30) than novices (mean = 1.71), suggesting that while professionals take more time, they make fewer mistakes. The results indicate that professional participants prioritize accuracy over speed, leading to a lower error rate but higher effort investment.

6.6. *TAM questionnaire results*

Fig. 12 presents an analysis of the TAM (Technology Acceptance Model) questionnaire to evaluate the ease of adopting SmellDSL. It is divided into three categories: *perceived ease of use*, *usefulness*, and *intention to use*. In the "Perceived ease of use" category, the majority of participating participants reported that the tool is easy to use (54% strongly agree), easy to learn (62%), and relatively easy to master (57%). As for "Perceived usefulness", 42% believe that SmellDSL facilitates program maintenance, while 34% believe that it helps with productivity. Furthermore, 45% strongly agree that it reduces the time to identify anomalies in the code. In the "Behavior intention" category, 58% said they would use SmellDSL as a support tool, showing a positive response to its use.

TAM data suggests that SmellDSL is well accepted and useful for specifying code smells, with positive intended use. The difference in performance between professionals and novices highlights the importance of experience in using technologies, reflecting strengths and challenges to consider when adopting the tool by different levels of users.

7. Study implications

This section brings some additional discussion by using SmellDSL to represent existing catalogs of bad smells from the literature (Section 7.1), introducing implications for both researchers (Section 7.2) and practitioners (Section 7.3), highlighting challenges for future research (Section 7.4), and outlining some limitations of our study (Section 7.5).

7.1. *SmellDSL and catalogs of bad smells*

In addition to the experimental study, we conducted a case study [87] to evaluate the expressiveness of the SmellDSL in specifying the catalogs of bad smells documented in the current literature [2,24,72,73]. One of our primary concerns was whether the SmellDSL could effectively represent the bad smell catalogs in the literature. If it could not, then its usefulness might be questionable. To address this, we conducted a case study to evaluate the capability of the SmellDSL to express these well-known bad smells. Following well-established guidelines [87] on organizing and conducting case studies, we structured our study to systematically assess whether SmellDSL could accurately capture the bad smell patterns documented in existing catalogs. This approach allowed us to test the language's expressiveness in practical contexts. All material produced can be found in our replication package [85].

For this, we collaboratively developed the bad smell pattern scripts. The first author initially created the scripts, while the others reviewed and provided feedback on the generated scripts. After two review cycles and iterative interactions among the authors, we agreed that the scripts accurately represented the cataloged bad smell patterns. This collaborative process ensured the correctness and completeness of the SmellDSL scripts in reflecting the known bad smells from the literature. In this sense, we perceived that the SmellDSL effectively represented the bad smells cataloged in the literature, demonstrating its expressiveness in capturing widely recognized patterns.

Moreover, we believe that these scripts may benefit the scientific community, mainly Ph.D. students, researchers, and practitioners, for two reasons. First, they offer a standardized, reusable code for specifying and analyzing bad smell patterns. Ph.D. students can streamline their investigations by reducing the effort to define these patterns manually. This allows researchers to focus on more advanced research, such as exploring new refactoring techniques or investigating the impact of bad smells on developers' cognitive load. Second, they enable automated detection of well-documented bad smells, improving code quality and reducing technical debt. Practitioners can easily modify and extend our scripts, making them more adaptable and effective in real-world development environments.

7.2. *Implications for researchers*

We present some research implications to pinpoint the potential influence that our findings can have on future research directions. It can guide the development of more robust bad smell specification approaches and applications. We also highlight areas where further investigation is needed to advance knowledge in the field.

Exploring human factors in emerging development environments. SmellDSL introduces a developer-sensitive approach to specifying bad smell patterns, which opens new avenues for investigating

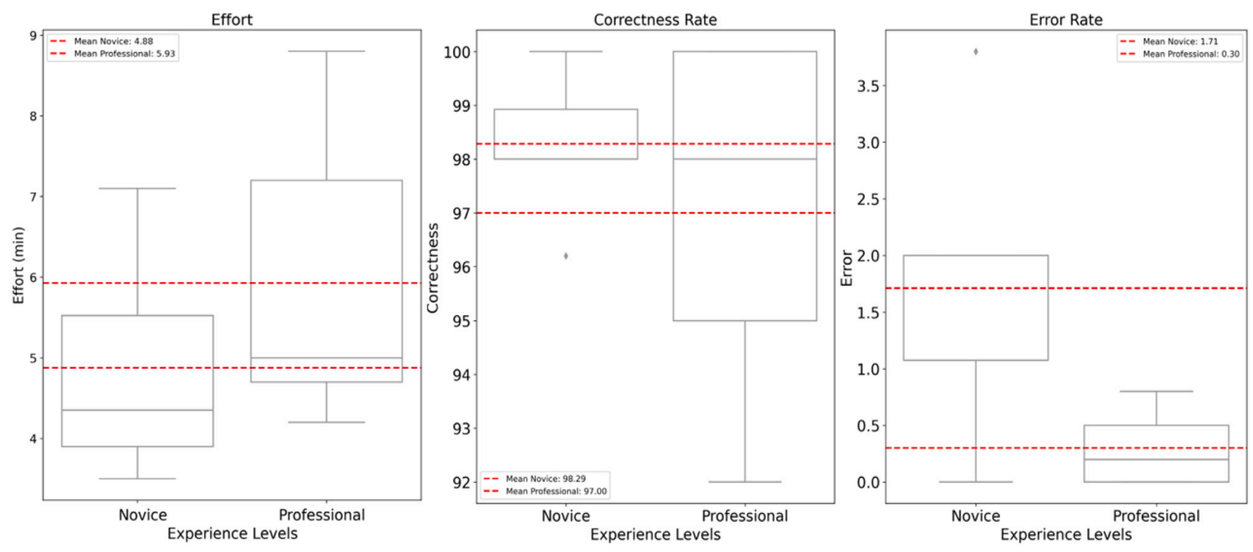


Fig. 11. Boxplots summarizing data produced by novices and professionals.

Table 10
Descriptive statistics of the results related to novice and professional.

Variable	Descriptive Statistics									Statistical Inference		
	Experience	N	Min	25th	Median	75th	Max	Mean	SD	t	D.F.	p-value
Effort	Novice	30	2	2.59	3.37	6.15	15.87	4.91	3.60	15	3.54	0.001
	Professional	5	2.37	2.50	5.75	9.43	12.37	5.92	4.04	15	3.61	0.001
Correctness Rate	Novice	30	79	100	100	100	100	98.30	4.97	0.05	4.88	0.001
	Professional	5	85	92.50	100	100	100	97	6.70	0.05	6	0.001
Error Rate	Novice	30	0	0	0	0	0.30	0.02	0.65	0.05	0.63	0.001
	Professional	5	0	0	0	0.07	0.15	0.03	0.06	0.05	0.06	0.001

Number of tests (N), Standard deviation (SD), 25th quartile (25th), 75th quartile (75th), Arithmetic mean (Mean), Maximum value (Max), Standard Deviation (SD), t-test value (t), Degrees of freedom (D.F), Statistical significance (p-value).

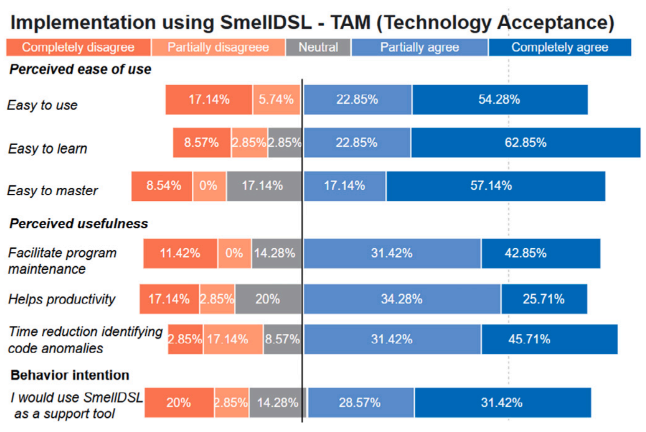


Fig. 12. Results of TAM questionnaire.

how developer perceptions influence software quality, including source code and architectural design quality. Based on our initial results, the research community could explore how tools like SmellDSL integrate into modern development environments such as Visual Studio Code, examining human factors like developer experience and ability, workflow integration, affective states [88], cognitive load [65,89] and use of biometric data [90]. Understanding how developers interact with these tools in real-world scenarios can help refine developer-centered DSLs and ensure their broad adoption. For instance, empirical studies comparing SmellDSL's user-friendliness and effectiveness in Visual Studio Code versus Eclipse could provide deeper insights into its practical benefits.

Robust empirical studies on scalability and usability. The initial success of SmellDSL in reducing effort and increasing correctness in code smell specification suggests promising results. However, new research is still needed to assess its scalability and robustness across diverse and complex projects. The research community might focus on conducting larger-scale empirical studies — mainly case studies to explore contextual and context phenomena [63,91] – that evaluate the tool's performance across various domains, programming languages, and team compositions. These studies could also investigate how SmellDSL supports evolving technologies like AI-assisted refactoring, combining human judgment with machine learning algorithms to optimize code smell detection while considering developers' preferences and reducing false positives.

Enhancing customizability in code smell detection tools. The SmellDSL study highlights the potential of customizable domain-specific languages in allowing developers to define and adapt code smells to their specific project needs. This opens up new research possibilities for enhancing the flexibility and adaptability of code smell detection tools. We believe that upcoming research could focus on how customizable DSLs like SmellDSL can be extended to accommodate evolving project requirements, programming languages, and frameworks. Additionally, studies could investigate how customizable tools impact long-term software maintenance and developer productivity, offering the research community valuable insights into balancing standardization and flexibility in automated software quality tools.

7.3. Implications for practitioners

SmellDSL to support collaborative specification and team dynamics: A novel approach. As software development increasingly relies on collaborative efforts across distributed teams [92,93], a key research challenge lies in SmellDSL to facilitate team-based decision-making and merging diverse perceptions of code quality. Investigating how SmellDSL can be used in collaborative development environments, such as Git-based workflows, could lead to tools that better accommodate varying developer expertise and preferences. Upcoming studies might explore how SmellDSL can improve the overall software quality in large, distributed projects, which usually adopt continuous delivery practices [94] and the use of biometric data to recommend bad smell specification conflicts [95]. Rubert and Farias [94] explored the effects of continuous delivery adoption on the source code and product quality through a case study in the industry. Dalcin et al. [95] have already indicated the potential of using biometric data and machine learning to recommend conflicts during software development processes. By supporting conflict resolution in code smells across different team members and enhancing team dynamics and communication, the SmellDSL might be adopted in the context of distributed bad smell specification.

Simplifying code smell specification in popular IDEs. SmellDSL offers developers a domain-specific language to customize code smell specifications, but its current integration is limited to Eclipse. Expanding support to more modern and popular platforms, such as Visual Studio Code, would increase its accessibility and adoption in day-to-day workflows. Developers using frameworks like Spring can leverage SmellDSL to fine-tune code smells specific to large enterprise applications. For example, allowing developers to define smells like Large Classes in Spring projects can lead to more focused and efficient refactoring strategies. For example, our study's participant (P4) reported that "*the greatest difficulty is understanding the combinations of smell features when specifying bad smell patterns. I had difficulty identifying how they relate to each other.*". Moreover, a forthcoming research venue might aim to explore recommendations on how to effectively mitigate the specified bad smell patterns, providing developers with actionable guidance for improving the specification of *treatments*. For example, the participant (P33) reported that given the bad smell at hand, he/she "had difficulty analyzing what to do to refactor the source code".

Improving software merge practices with developer-sensitive specification and detection. The ability to specify developer-sensitive code smells can help mitigate risks in complex merge scenarios of bad smell specification code, where differing team perceptions of code quality often result in conflicts. SmellDSL, with its pivotal role in providing a shared framework for specifying code smells, can help teams align their understanding concerning bad smell patterns, reducing complex specification conflicts. Integrating biometric feedback (such as cognitive load metrics [89]) into SmellDSL could further support developers by identifying when the complexity of code changes increases cognitive stress, allowing for real-time adjustments in the specification of smells that are critical during merges.

7.4. Challenges for future research

This section outlines critical future research challenges arising from using domain-specific languages for bad smell specification, highlighting areas that require further investigation to improve specification and detection accuracy, reduce cognitive load, and enhance tool adaptability in real-world software development environments.

Empirical evaluation of machine learning techniques for code smell classification using EEG data. Integrating biometric data, such as EEG signals, presents an opportunity for enhancing code smell detection through machine learning techniques. Developers could benefit from tools that empirically evaluate and classify bad smell patterns based on cognitive responses during code review and refactoring tasks. For instance, training machine learning models on EEG data could help identify moments of high cognitive load, correlating these with specific types of bad smells. Previous studies have reported empirical evaluations of machine learning techniques to classify source code comprehension based on EEG data, exploring the impact of modularization on cognitive load. A similar investigation regarding harmful bad smell specifications could allow developers to focus on technically and cognitively challenging areas, significantly improving overall productivity and code quality in environments like Visual Studio Code or during complex merges.

Reducing cognitive load in real-time code smell detection. Another challenge is reducing cognitive load during real-time bad smell detection in integrated development environments (IDEs) like Visual Studio Code. As developers manage complex tasks like software merging, the simultaneous identification and resolution of bad smells can overwhelm cognitive resources. For example, in high-stress scenarios where a merge conflict arises due to inconsistent code smells, using biometric data (such as EEG readings) could inform automated tools to adjust the specificity of smell detection in real-time, lowering the cognitive burden on the developer. This challenge involves designing empirical studies that explore the relationship between cognitive load and the complexity of bad smell patterns.

Empirical studies on specification and detection of bad smell patterns. There is a critical need for empirical studies focused on specifying and detecting bad smell patterns within software engineering. This research is of utmost importance, as while significant contributions to the field have primarily centered on code comprehension, little empirical knowledge exists regarding developers' cognitive processes when identifying bad smells. Key areas for investigation include the impact of task difficulty and code complexity on cognitive load, the stress levels associated with refactoring, and how interruptions affect the detection of code smells. We believe that EEG devices (*e.g.*, Emotiv EPOC[7]) and eye trackers (*e.g.*, Tobii[8]) can provide valuable insights into developers' cognitive processes and emotional states, helping to address this challenge by capturing real-time data on brain activity and visual attention during the specification and detection of bad smell patterns. Building on foundational studies such as [96,97], future research should comprehensively structure controlled experiments to explore these dynamics. The potential impact of this research can be significant, as understanding how cognitive load fluctuates with increasing code complexity can inform better tooling and methodologies for bad smell detection. Additionally, investigating developers' brain functions during these tasks could reveal insights into how cognitive indicators relate to identifying and resolving bad smells. Addressing this challenge can enhance our understanding of the cognitive factors influencing software development and support the creation of more effective tools for managing code quality.

7.5. Study limitations

Despite the promising results, this study has several limitations that must be acknowledged. First, SmellDSL was evaluated using a limited set of code smells (only eight scenarios) from existing catalogs, which may only partially represent the complexity and diversity of real-world software projects. While the selected bad smells are commonly cited in the literature, the applicability of SmellDSL to detect less frequent or more context-specific code smells still needs to be tested. Additionally, the study focused on the tool's ability to specify known bad smells, leaving its performance in detecting new or emerging patterns unexplored.

Another limitation concerns the scope of the empirical evaluation, which involved a relatively small group of participants and a controlled experimental environment. This restricts the generalizability

[7] Wireless EEG Headset: https://www.emotiv.com/

[8] Tobii Eye-Tracker: https://www.tobii.com/

of the results, as the tool's effectiveness and usability may vary significantly across larger teams, diverse project settings, or different development environments. Furthermore, the study did not thoroughly compare SmellDSL with fully automated tools like deep learning-based approaches, which limits our understanding of its relative performance in automated detection tasks. These factors highlight the need for future research to address broader contexts and larger, more diverse datasets.

Additional limitations include the participants' inability to automate the generation of bad smell quantities and the tool's integration being limited to the Eclipse platform. Some participants suggested that it would be interesting to visualize the execution of SmellDSL specifications in a debug mode, in which they might grasp the excitation of the SmellDSL rules to specify the bad smell patterns. The language also suffers from limited documentation and examples for developers, and the capacity to express symptoms is restricted to 500 characters. Furthermore, SmellDSL has not yet been compared with other tools, though it does not impose any restrictions for such comparisons. Future work will evaluate SmellDSL against other existing tools.

8. Threats to validity

Our study can have threats to validity that range from internal, construct, and statistical conclusion validity threats to external threats. We discuss how these threats were mitigated.

8.1. Statistical conclusion validity

We minimized this threat by ensuring that the independent and dependent variables were subject to appropriate statistical methods [62]. The evaluation addressed two key concerns: (1) determining if there is a correlation between cause and effect and (2) evaluating the strength of this relationship. These checks help avoid incorrect conclusions about the existence or nonexistence of causal relationships between the variables. Additionally, the analysis examined how confidently we can estimate the strength of this covariation. To address the covariance between cause and effect, we analyzed the normal distribution of the collected sample to ensure appropriate statistical methods were applied. The Kolmogorov–Smirnov and Shapiro–Wilk tests were used to verify the data's normality, ensuring that parametric or non-parametric statistical methods were used correctly. We tested all hypotheses at a significance level of 0.05. To increase the power of the analysis, we gathered a more significant number of experimental cases, improving the reliability of our results. Additionally, the participants used real-world scenarios, reducing the risk of errors that could obscure the relationships between variables. Consequently, this approach ensured the reliability and confidence of the findings in this study.

8.2. Construct validity

It refers to the extent to which the operational measures used in the study accurately capture the theoretical constructs they are intended to represent [62]. In this study, we evaluated (1) whether the methods for quantifying the dependent variables were appropriate, (2) whether the quantification was performed accurately, and (3) whether any threats to validity arose from the nature of the specifications being analyzed.

Quantification methods. The concept of *effort* used in our study is a well-established construct in software engineering research. We adopted the quantification method from prior studies [63,64,98,99]. We measured effort based on the time (in minutes) that participants invested in each code smell specification task. We employed a suite of pre-defined metrics that were independently validated in previous studies for correctness and error rate. These metrics helped ensure we could measure the degree to which SmellDSL specifications conformed to the predefined code smell rules.

Ensuring quantification accuracy. The authors collaborated closely to ensure that the quantification process was accurate. Data collection adhered to guidelines and frameworks established in earlier research [64,100], ensuring that the measurement of variables such as effort, correctness, and error rate aligned with the study's objectives and hypotheses. The procedure was thoroughly planned and executed to mitigate any errors in data handling.

Execution and tooling effects. The use of manual and semi-automated techniques during specification tasks was controlled to avoid unintentional bias in the results. While manual processes were useful for minimizing tool-related issues, we were cautious about the potential influence of using the Eclipse platform and SmellDSL's specific features. Although tool limitations could have affected participant performance, precautions were taken to ensure that tasks remained tool-agnostic, limiting any possible impact on the study's outcomes. Consequently, we believe that the design and use of tools did not significantly threaten the internal validity of the experimental results.

8.3. Internal validity

Inferences between our independent variable (the use of SmellDSL) and the dependent variables (effort, correctness rate, and error rate) are considered internally valid if a causal relationship can be established. Our study meets this criterion because: (1) temporal precedence was maintained, as the specification of bad smells occurred before the measurement of errors and effort; (2) covariation was observed, indicating that the use of SmellDSL influenced both effort and correctness; and (3) no external factors were identified that could influence this covariation. Thus, our study satisfies all three requirements for internal validity. We further supported internal validity through additional analyses. Specific cases were conducted to demonstrate that the independent variable exclusively affected the dependent variables. Our observations confirmed that modifications to SmellDSL scripts directly impacted the recorded values of the controlled variables. However, we did identify some threats to internal validity. First, because some measures of the dependent variables were calculated manually, there was a risk of data reliability issues, which could lead to inconsistent results. We mitigated this risk by establishing clear measurement guidelines and conducting two rounds of data reviews with the authors.

Another threat to internal validity is the presence of confounding variables, which can unexpectedly influence the dependent variables. To address this, we conducted a pilot study to ensure that the dependent variables were solely affected by the use of SmellDSL. During this pilot, we identified potential confounding factors, such as the size of bad smell descriptions, that could impact the outcomes.

8.4. External validity

External validity pertains to the applicability of our findings to broader contexts. Specifically, it examines the extent to which the results of this study on SmellDSL can be generalized to other scenarios, such as different code smell specifications, varying model sizes, or developers with diverse levels of experience. We analyzed whether the causal relationships identified could hold true across variations in participants, tools, and environments. Since this study has not yet been replicated, we utilized Donald T. Campbell's theory [101] of proximal similarity to assess the generalization of our results. This approach helps define criteria for identifying contexts where our findings may apply.

The criteria we considered include the following: First, developers should be able to utilize a DSL to specify bad smell patterns. Second, these specifications should apply to various types of bad smells. It is important to note that the specifications used in our study were relatively small. Finally, developers should use the SmellDSL. Given that these criteria are relevant in mainstream bad smell specification, we conclude that the results of our study may be applicable, at least partially, to similar contexts that meet these conditions.

9. Conclusions and future work

This article introduced SmellDSL, a domain-specific language designed to assist developers in specifying code smells. We conducted an experimental study to assess the impact of SmellDSL on three key variables: the effort invested in specifying code smells, the correctness rate of such specifications, and the error rate. The study yielded impressive results, further validating the effectiveness and reliability of SmellDSL. The primary focus of our experiments was on measuring the correctness rate and task completion times, and we did not perform a comparative analysis of false positive rates with other methods. To address this gap, we plan to incorporate a dedicated analysis of false positives in future studies. This includes direct comparisons to established methods and a discussion of the mechanisms within SmellDSL aimed at reducing false positives.

The primary outcome of this research was the practical application of SmellDSL, which significantly streamlines the process of identifying and addressing developer-sensitive code smells, making it a routine task for developers. By leveraging SmellDSL, developers can also identify refactoring opportunities related to symptoms of architectural degradation. The results were promising, with a correctness rate exceeding 88%, an error rate below 12%, and a specification time of less than 5 min for a code smell. While these findings are encouraging, further experiments are needed to evaluate the language benefits in a broader scenario.

In future work, we focus on (1) developing a simplified textual notation for bad smell specifications and integrating checkers into the Eclipse IDE with inline violation messages, thereby providing a complete development environment for users; (2) in future steps, we plan to extend SmellDSL to other IDEs, including Visual Studio Code, to enhance its accessibility and utility (3) additionally, we will outline potential improvements for enhancing usability, such as refining the interface, expanding documentation, and incorporating interactive help features (4) creating a SmellDSL-Engine framework that utilizes a predefined metrics catalog. The framework will accept scripts generated by SmellDSL as input parameters, enabling the quantification of previously established metrics, which can be extended through XML definitions. Metrics will be calculated, along with the evaluation of rules created by SmellDSL. This research can be seen as the first step in a comprehensive schedule that provides developers with the resources to specify code smells effectively using pattern models (5) we plan to outline our vision for future evaluations that will investigate how SmellDSL compares with traditional metric-based methods in handling subjectivity and exploring more comprehensively the features within SmellDSL that are intended to align with developers' experiences and perceptions.

CRediT authorship contribution statement

Robson Keemps: Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Methodology, Formal analysis, Conceptualization. **Kleinner Farias:** Writing – review & editing, Writing – original draft, Validation, Supervision, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Rafael Kunst:** Writing – review & editing, Writing – original draft, Validation, Supervision, Methodology, Investigation, Conceptualization. **Carlos Carbonera:** Writing – review & editing, Writing – original draft, Validation, Supervision, Methodology, Investigation, Formal analysis, Conceptualization. **Willian Bolzan:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Methodology, Formal analysis, Conceptualization.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Kleinner Silva Farias de Oliveira reports financial support was provided by Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq). Kleinner Silva Farias de Oliveira reports financial support was provided by Fundaçã o de Ampara a Pesquisa do Estado do Rio Grande do Sul (FAPERGS). Robson Keemps da Silva reports financial support was provided by Coordenacã o de Aperfeiçoamento de Pessoal de Nível Superior (CAPES). If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported in part by the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), Brazil, under Grant No. 314248/2021-8, and Fundação de Amparo à Pesquisa do Estado do Rio Grande do Sul (FAPERGS), Brazil, under Grant No. 21/2551-0002051-2. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 88887.897203/2023-00.

Data availability

Data will be made available on request.

References

[1] Willian Oizumi, et al., Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems, in: 38th International Conference on Software Engineering, 2016, pp. 440–451.
[2] Girish Suryanarayana, Ganesh Samarthyam, Tushar Sharma, Refactoring for Software Design Smells: Managing Technical Debt, Morgan Kaufmann, 2014.
[3] Eman Abdullah AlOmar, et al., On preserving the behavior in software refactoring: A systematic mapping study, Inf. Softw. Technol. 140 (2021) 106675.
[4] Eman Abdullah AlOmar, et al., Behind the intent of extract method refactoring: A systematic literature review, IEEE Trans. Softw. Eng. (2024).
[5] Nikolaos Nikolaidis, et al., A metrics-based approach for selecting among various refactoring candidates, Empir. Softw. Eng. 29 (1) (2024) 25.
[6] Nikolaos Tsantalis, Ameya Ketkar, Danny Dig, RefactoringMiner 2.0, IEEE Trans. Softw. Eng. 48 (3) (2020) 930–950.
[7] Morteza Zakeri-Nasrabadi, et al., A systematic literature review on the code smells datasets and validation mechanisms, ACM Comput. Surv. 55 (13s) (2023) 1–48.
[8] Daniel Oliveira, et al., Developers' perception matters: machine learning to detect developer-sensitive smells, Empir. Softw. Eng. 27 (7) (2022).
[9] Leonardo Sousa, et al., Identifying design problems in the source code: A grounded theory, in: 40th International Conference on Software Engineering, 2018, pp. 921–931.
[10] Chunhao Dong, Yanjie Jiang, Nan Niu, Yuxia Zhang, Hui Liu, Context-aware name recommendation for field renaming, in: IEEE Transactions on Software Engineering, 2024, pp. 1–13.
[11] Sharanpreet Kaur, Satwinder Singh, Object oriented metrics based empirical model for predicting "code smells" in open source software, JIE (India): Ser. B 104 (1) (2023) 241–257.
[12] Lech Madeyski, Tomasz Lewowski, Detecting code smells using industry-relevant data, Inf. Softw. Technol. 155 (2023) 107112.
[13] Bigonha Mariza, et al., The usefulness of software metric thresholds for detection of bad smells and fault prediction, Inf. Softw. Technol. 115 (2019) 79–92.
[14] Somayeh Kalhor, et al., A systematic review of refactoring opportunities by software antipattern detection, Autom. Softw. Eng. 31 (2) (2024) 1–65.
[15] Fabio Palomba, Damian Andrew Tamburri, Predicting the emergence of community smells using socio-technical metrics: A machine-learning approach, J. Syst. Softw. 171 (2021) 110847.
[16] Amjed Tahir, et al., A large scale study on how developers discuss code smells and anti-pattern in stack exchange sites, Inf. Softw. Technol. 125 (2020) 106333.
[17] Mario Hozano, et al., Are you smelling it? Investigating how similar developers detect code smells, Inf. Softw. Technol. 93 (C) (2018) 130—-146.
[18] L. Sousa, W. Oizumi, A. Garcia, A. Oliveira, D. Cedrim, C. Lucena, When are smells indicators of architectural refactoring opportunities? a study of 50 software projects, in: International Conference on Program Comprehension, ICPC, IEEE, 2020, pp. 354–365.
[19] Xiaofeng Han, et al., Code smells detection via modern code review: A study of the openstack and qt communities, Empir. Softw. Eng. 27 (6) (2022) 127.

[20] Ed Wilson Júnior, Kleinner Farias, Bruno da Silva, On the use of uml in the brazilian industry: A survey, J. Softw. Eng. Res. Dev. 10 (2022).
[21] Guilherme Lacerda, Fabio Petrillo, Marcelo Pimenta, Yann Gaël Guéhéneuc, Code smells and refactoring: A tertiary systematic review of challenges and observations, J. Syst. Softw. 167 (2020) 110610.
[22] José Pereira, et al., Code smells detection and visualization: a systematic literature review, Arch. Comput. Methods Eng. 29 (1) (2022) 47–94.
[23] R. Marinescu, Measurement and quality in object-oriented design, in: 21st IEEE International Conference on Software Maintenance, ICSM'05, 2005, pp. 701–704.
[24] Michele Lanza, Radu Marinescu, Object-Oriented Metrics in Practice. Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-oriented Systems, Springer Verlag, 2010.
[25] Daniel Coutinho, et al., On the influential interactive factors on degrees of design decay: A multi-project study, in: IEEE Transactions on Software Engineering, 2022, pp. 753–764.
[26] Oussama Sghaier, et al., Fighting evil is not enough when refactoring metamodels: promoting the good also matters, in: 37th ACM/SIGAPP Symposium on Applied Computing, SAC '22, Association for Computing Machinery, 2022, pp. 1517–1526.
[27] Jairo Souza, et al., Developers' viewpoints to avoid bug-introducing changes, Inf. Softw. Technol. 143 (2022) 106766.
[28] Danyllo Albuquerque, et al., Integrating interactive detection of code smells into scrum: Feasibility, benefits, and challenges, Appl. Sci. 13 (15) (2023) 8770.
[29] Anderson Oliveira, et al., Smell patterns as indicators of design degradation: Do developers agree? in: XXXVI Brazilian Symposium on Software Engineering, SBES, 2022, pp. 311–320.
[30] Anderson Uchôa, Unveiling multiple facets of design degradation in modern code review, in: 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, in: ESEC/FSE 2021, Association for Computing Machinery, New York, NY, USA, 2021, pp. 1615–1619.
[31] André Eposhi, et al., Removal of design problems through refactorings: Are we looking at the right symptoms? in: International Conference on Program Comprehension, ICPC, IEEE, 2019, pp. 148–153.
[32] Júlio Martins, et al., Are code smell co-occurrences harmful to internal quality attributes? A mixed-method study, in: Brazilian Symposium on Software Engineering, SBES '20, 2020, pp. 52–61.
[33] Willian Oizumi, Leonardo Sousa, Anderson Oliveira, Luiz Carvalho, Alessandro Garcia, Thelma Colanzi, Roberto Oliveira, On the density and diversity of degradation symptoms in refactored classes: A multi-case study, in: 30th International Symposium on Software Reliability Engineering, ISSRE, 2019, pp. 346–357.
[34] Anderson Uchôa, et al., How does modern code review impact software design degradation? an in-depth empirical study, in: International Conference on Software Maintenance and Evolution, ICSME, IEEE, 2020, pp. 511–522.
[35] Marc Eaddy, Thomas Zimmermann, Kaitlin D. Sherwood, Vibhav Garg, Gail C. Murphy, Nachiappan Nagappan, Alfred V. Aho, Do crosscutting concerns cause defects? IEEE Trans. Softw. Eng. 34 (4) (2008) 497–515.
[36] Eduardo Figueiredo, et al., On the Maintainability of Aspect-Oriented Software: A Concern-Oriented Measurement Framework, IEEE, 2008, pp. 183–192.
[37] Juliana Padilha, et al., On the Effectiveness of Concern Metrics to Detect Code Smells: An Empirical Study, Springer, 2014, pp. 656–671.
[38] D. Jackson, The Essence of Software: Why Concepts Matter for Great Design, Princeton University Press, 2021.
[39] Roger S. Pressman, Software Engineering: A Practitioner's Approach, Pressman and Associates, 2005.
[40] Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, Anne-Francoise Le Meur, DECOR: A method for the specification and detection of code and design smells, IEEE Trans. Softw. Eng. 36 (1) (2010) 20–36.
[41] Chidamber, Kemerer, A metrics suite for object oriented design, IEEE Trans. Softw. Eng. 20 (6) (1994) 476–493.
[42] Lorenzo Bettini, et al., Supporting safe metamodel evolution with edelta, Softw. Tools Technol. Transf. 24 (2) (2022) 247–260.
[43] Kostadin Rajkovic, Eduard Enoiu, Nalabs: Detecting bad smells in natural language requirements and test specifications, 2022, arXiv preprint arXiv:2202.05641.
[44] Angela Barriga, et al., Addressing the trade off between smells and quality when refactoring class diagrams, J. Object Technol. 20 (3) (2021).
[45] Naouel Moha, et al., From a domain analysis to the specification and detection of code and design smells, Form. Asp. Comput. 22 (3) (2010) 345–361.
[46] Luis Felipi Junionello, et al., Revealing developers' arguments on validating the incidence of code smells: A focus group experience, in: IX Workshop on Visualization, Evolution and Maintenance of Software, SBC, 2021, pp. 31–35.
[47] Aiko Yamashita, et al., Inter-smell relations in industrial and open source systems: A replication and comparative analysis, in: International Conference on Software Maintenance and Evolution, ICSME, IEEE, 2015, pp. 121–130.
[48] Amal Alazba, Hamoud Aljamaan, Code smell detection using feature selection and stacking ensemble: An empirical investigation, Inf. Softw. Technol. 138 (2021) 106648.
[49] Kleinner Farias, Luan Lazzari, Event-driven architecture and REST architectural style: An exploratory study on modularity, J. Appl. Res. Technol. (2023).
[50] Roger Denis Vieira, Kleinner Farias, Usage of psychophysiological data as an improvement in the context of software engineering: A systematic mapping study, in: XVI Brazilian Symposium on Information Systems, SBSI '20, Association for Computing Machinery, 2020.
[51] Muhammad Azeem, et al., Machine learning techniques for code smell detection: A systematic literature review and meta-analysis, Inf. Softw. Technol. 108 (2019) 115–138.
[52] Eclipse Foundation, Xtext - Language engineering made easy, 2024, URL https://eclipse.dev/Xtext/. (Accessed 30 April 2024).
[53] Moritz Eysholdt, Heiko Behrens, Xtext: implement your language faster than the quick and dirty way, in: ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion, 2010, pp. 307–309.
[54] Lorenzo Bettini, Implementing Domain-Specific Languages with Xtext and Xtend, Packt Publishing Ltd, 2016.
[55] Alexandre Bragança, et al., Towards supporting SPL engineering in low-code platforms using a DSL approach, 2021, pp. 16–28.
[56] Wolf Rost, Mining of DSLs and generator templates from reference applications, in: IEEE Transactions on Software Engineering, 2020, pp. 1–7.
[57] Younes Boubekeur, et al., A DSL and model transformations to specify learning corpora for modeling assistants, in: 25th International Conference on Model Driven Engineering Languages and Systems, 2022, pp. 95–102.
[58] Jose Pablo De La Rosa, et al., End-user programming of robot-assisted physical training activities through behaviour-driven development, in: IEEE Transactions on Software Engineering, 2024, pp. 387–391.
[59] Yinling Liu, Jean-Michel Bruel, Modeling and verification of natural language requirements based on states and modes, Form. Asp. Comput. (2024).
[60] Marcel van Amstel, Mark van den Brand, Luc Engelen, An exercise in iterative domain-specific language design, in: The Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution, IWPSE, 2010, pp. 48–57.
[61] Weixing Zhang, Jan-Philipp Steghöfer, Regina Hebig, Daniel Strüber, A rapid prototyping language workbench for textual DSLs based on Xtext: Vision and progress, 2023, arXiv arXiv:2309.04347.
[62] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, Anders Wesslén, Experimentation in Software Engineering, Springer, 2012.
[63] Leandro Ferreira D'Avila, Kleinner Farias, Jorge Luis Victória Barbosa, Effects of contextual information on maintenance effort: A controlled experiment, J. Syst. Softw. 159 (2020) 110443.
[64] Kleinner Farias, et al., Evaluating the effort of composing design models: A controlled experiment, Softw. Syst. Model. 14 (2015) 1349–1365.
[65] Matheus Segalotto, Willian Bolzan, Kleinner Farias, Effects of modularization on developers' cognitive effort in code comprehension tasks: A controlled experiment, in: Brazilian Symposium on Software Engineering, 2023, pp. 206–215.
[66] Fabio Palomba, Annibale Panichella, Andy Zaidman, Rocco Oliveto, Andrea De Lucia, The scent of a smell: An extensive comparison between textual and structural smells, in: 40th International Conference on Software Engineering, 2018, pp. 740–740.
[67] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, Denys Poshyvanyk, When and why your code starts to smell bad (and whether the smells go away), IEEE Trans. Softw. Eng. 43 (11) (2017) 1063–1088.
[68] Kleinner Farias, Alessandro Garcia, Carlos Lucena, Effects of stability on model composition effort: an exploratory study, Softw. Syst. Model. 13 (4) (2014) 1473–1494.
[69] Claes Wohlin, Case study research in software engineering—It is a case, and it is a study, but is it a case study? Inf. Softw. Technol. 133 (2021) 106514.
[70] Willian Oizumi, Leonardo Sousa, Alessandro Garcia, Roberto Oliveira, Anderson Oliveira, OI Anne Benedicte Agbachi, Carlos Lucena, Revealing design problems in stinky code: a mixed-method study, in: 11th Brazilian Symposium on Software Components, Architectures, and Reuse, 2017, pp. 1–10.
[71] Nikola Marangunić, Andrina Granić, Technology acceptance model: A literature review from 1986 to 2013, Univers. Access Soc. 14 (2015) 81–95.
[72] Khalid Alkharabsheh, et al., Software Design Smell Detection: a systematic mapping study, Softw. Qual. J. 27 (3) (2019) 1069–1148.
[73] Martin Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley Longman Publishing Co. Inc., USA, 1999.
[74] Danyllo Albuquerque, et al., On the assessment of interactive detection of code smells in practice: A controlled experiment, IEEE Trans. Softw. Eng. 11 (2023) 84589–84606.
[75] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, Dave Binkley, Are test smells really harmful? an empirical study, Empir. Softw. Eng. 20 (2015) 1052–1094.
[76] Roberto Oliveira, et al., Collaborative or individual identification of code smells? On the effectiveness of novice and professional developers, Inf. Softw. Technol. (2020).

[77] Md Masudur Rahman, Abdus Satter, Md Mahbubul Alam Joarder, Kazi Sakib, An empirical study on the occurrences of code smells in open source and industrial projects, in: 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, 2022, pp. 289–294.

[78] Mohammed Latif Siddiq, Shafayat H Majumder, Maisha R Mim, Sourov Jajodia, Joanna CS Santos, An empirical study of code smells in transformer-based code generation techniques, in: 22nd International Working Conference on Source Code Analysis and Manipulation, SCAM, IEEE, 2022, pp. 71–82.

[79] Riya Dutta, Diego Elias Costa, Emad Shihab, Tanja Tajmel, Diversity awareness in software engineering participant research, in: 45th International Conference on Software Engineering, IEEE, 2023, pp. 120–131.

[80] Valentina Lenarduzzi, Oscar Dieste, Davide Fucci, Sira Vegas, Towards a methodology for participant selection in software engineering experiments: A vision of the future, in: 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM, 2021, pp. 1–6.

[81] Austen Rainer, Claes Wohlin, Recruiting credible participants for field studies in software engineering research, Inf. Softw. Technol. 151 (2022) 107002.

[82] Davide Falessi, et al., Empirical software engineering experts on the use of students and professionals in experiments, ESE 23 (2018) 452–489.

[83] Iflaah Salman, Ayse Tosun Misirli, Natalia Juristo, Are students representatives of professionals in software engineering experiments? in: IEEE/ACM 37th International Conference on Software Engineering, vol. 1, 2015, pp. 666–676.

[84] Heather Wild, Aki-Juhani Kyröläinen, Victor Kuperman, How representative are student convenience samples? A study of literacy and numeracy skills in 32 countries, Plos One 17 (7) (2022) e0271191.

[85] SmellDSL, Smelldsl: A domain-specific language, 2024, http://dx.doi.org/10.5281/zenodo.13916660, URL https://github.com/kleinnerfarias/smelldsl. (Accessed 10 October 2024).

[86] Filippo Ricca, et al., How developers' experience and ability influence web application comprehension tasks supported by uml stereotypes: A series of four experiments, IEEE Trans. Softw. Eng. 36 (1) (2009) 96–118.

[87] Per Runeson, Martin Höst, Guidelines for conducting and reporting case study research in software engineering, Empir. Softw. Eng. 14 (2009) 131–164.

[88] Mateus Manica, Kleinner Farias, Lucian Gonçales, Vinícius Bischoff, Bruno Carreiro da Silva, Everton T Guimarães, Effects of model composition techniques on effort and affective states: A controlled experiment (S), in: SEKE, 2018, pp. 304–303.

[89] Lucian José Gonçales, Kleinner Farias, Bruno C da Silva, Measuring the cognitive load of software developers: An extended Systematic Mapping Study, Inf. Softw. Technol. 136 (2021) 106563.

[90] Juliano Paulo Menzen, Kleinner Farias, Vinicius Bischoff, Using biometric data in software engineering: a systematic mapping study, Behav. Inf. Technol. 40 (9) (2021) 880–902.

[91] Leandro D'Avila, et al., SW-Context: a model to improve developers' situational awareness, IET Softw. 14 (5) (2020) 535–543.

[92] Carlos Carbonera, et al., Software merge: A two-decade systematic mapping study, in: Brazilian Symposium on Software Engineering, 2023, pp. 99–108.

[93] César Augusto Graeff, Kleinner Farias, Carlos Eduardo Carbonera, On the prediction of software merge conflicts: A systematic review and meta-analysis, in: XIX Brazilian Symposium on Information Systems, 2023, pp. 404–411.

[94] Maluane Rubert, Kleinner Farias, On the effects of continuous delivery on code quality: A case study in industry, Comput. Stand. Interfaces 81 (2022) 103588.

[95] Guilherme Dalcin, et al., Recommendation of UML model conflicts: Unveiling the biometric lens for conflict resolution, in: Brazilian Symposium on Software Engineering, 2023, pp. 83–88.

[96] Norman Peitek, A Neuro-Cognitive Perspective of Program Comprehension (Ph.D. thesis), Technische Universität Chemnitz, 2022.

[97] Janet Siegmund, Norman Peitek, André Brechmann, Chris Parnin, Sven Apel, Studying programming in the neuroage: just a crazy idea? Commun. ACM 63 (6) (2020) 30–34.

[98] Carlos Eduardo Carbonera, Kleinner Farias, Vinicius Bischoff, Software development effort estimation: A systematic mapping study, IET Softw. 14 (4) (2020) 328–344.

[99] Kleinner Farias, Empirical evaluation of effort on composing design models, 2016, arXiv preprint arXiv:1610.09012.

[100] Kleinner Farias, Alessandro Garcia, Jon Whittle, Carlos Lucena, Analyzing the effort of composing design models of large-scale software in industrial case studies, in: Model-Driven Engineering Languages and Systems, Springer, 2013, pp. 639–655.

[101] Donald Thomas Campbell, M. Jean Russo, Social experimentation, SAGEClassics, Beverly Hills, 1999.