Code Smells in Functional Programming: Initial Findings from a Grey Literature Review on Clojure

Walber Araújo
Federal University of Campina Grande (UFCG)
Campina Grande, Brazil
walber.araujo@splab.ufcg.edu.br

José Truta
Federal University of Campina Grande (UFCG)
Campina Grande, Brazil
jose.truta@splab.ufcg.edu.br

Lucas Vegi
Federal University of Viçosa (UFV)
Viçosa, Brazil
lucas.vegi@ufv.br

Marco Tulio Valente
Federal University of Minas Gerais (UFMG)
Belo Horizonte, Brazil
mtov@dcc.ufmg.br

João Brunet
Federal University of Campina Grande (UFCG)
Campina Grande, Brazil
joao.arthur@computacao.ufcg.edu.br

Abstract

Code smells are widely used indicators of poor code quality, revealing structural problems and areas where improvement can be made. Although extensively studied in object-oriented languages, functional programming languages remain comparatively underexplored in literature. This paper presents early results from a grey literature investigation of code smells in Clojure, a modern functional programming language that runs on the Java Virtual Machine (JVM) and is widely adopted in open source and industrial systems. Inspired by prior work on Elixir, we manually inspected developer discussions retrieved through Google search, extracted quality concerns discussed by developers, and had 44 practitioners evaluate the relevance of non-traditional smell candidates. As preliminary results, we cataloged 26 code smells, including 12 Clojure-specific, 9 functional-style, and 5 traditional Fowler smells. We also analyzed tool support and observed a significant gap, as existing Clojure linters cover only 2 of these 26 smells. These findings provide an initial characterization of how Clojure developers discuss code smells, a preliminary set of smell-like problems specific to this ecosystem, and an early assessment of tool support for their detection.

Keywords

Code Smells, Clojure, Grey Literature

1 Introduction

As software systems evolve, ensuring quality aspects that positively influence maintainability becomes increasingly important, especially because development costs are typically associated with maintenance activities [18, 23]. Poor design or implementation decisions may introduce recurring symptoms of degraded quality, commonly referred to as code smells, which may reveal deeper problems and indicate opportunities for refactoring [12]. Although they do not necessarily affect program execution, such problems may hinder maintenance and software evolution over time. The foundational catalog by Fowler and Beck has motivated extensive research on code smells across diverse domains, from mobile applications [5] to machine learning systems [32]. These studies demonstrate that smells establish a shared vocabulary for quality issues, directly influencing the development of static analysis tools like SonarQube [26] and PMD [24] to automatically detect such problems.

While code smells have been widely studied, functional programming languages remain comparatively underexplored in this literature. Prior work on Elixir provided early evidence that developers discuss both traditional smells and language-specific quality concerns [29]. This line of research was later extended into broader investigations of Elixir-specific code smells and refactorings, validated with developers from the community [28, 30]. However, these findings are still limited to a single ecosystem. Moreover, the authors explicitly point to the investigation of other modern functional languages, such as Clojure [16], as a direction for future work. Motivated by this gap, we replicate the original study in the Clojure ecosystem to investigate whether similar discussions about code smells also emerge in this context.

In this paper, we present early results from a grey literature investigation of code smells in Clojure, a functional programming language that runs on the Java Virtual Machine (JVM) and is widely adopted in industrial systems by companies such as Apple and Nubank[1]. Besides its industrial relevance, Clojure combines features such as macros, lazy evaluation, immutable data structures, software transactional memory, and seamless interoperability with the Java ecosystem[2], enabling developers to combine functional programming with object-oriented APIs within the same codebase. These characteristics contribute to a distinctive programming model that may give rise to language-specific maintainability concerns not reported in studies of other programming languages. Despite this, no previous study has systematically investigated code smells in the Clojure ecosystem. Inspired by previous work on Elixir, we investigate whether Clojure developers discuss traditional smells, identify language-specific quality concerns, and evaluate whether such issues are supported by well-known static analysis tools. Since this study focuses on a single source of grey literature, our goal is not to provide a validated catalog, but to report preliminary evidence on smells and related quality concerns in developer discussions.

To conduct our replication, we adapted the original research questions to focus on the Clojure ecosystem:

[1] https://clojure.org/community/companies
[2] https://clojure.org/reference/java_interop

- RQ1. Do *Clojure* developers discuss traditional code smells?
- RQ2. Do *Clojure* developers discuss other smells?
- RQ3. Do well-known static code analysis tools for Clojure detect code smells?

To answer these questions, we manually inspected the retrieved grey literature documents, selected those relevant to our research questions, and extracted the smells discussed by developers. We then distinguished traditional smells from non-traditional candidates treated as recurring quality concerns in this ecosystem. After an initial refinement step, practitioners evaluated the relevance of these candidates based on their names and descriptions. Finally, we examined whether well-known static analysis tools detect them.

The main contributions of this paper are: (i) an initial characterization of how Clojure developers discuss traditional code smells in grey literature; (ii) a set of smell-like problems discussed in the Clojure ecosystem; and (iii) an assessment of the extent to which Clojure static analysis tools support their detection.

The remainder of this paper is organized as follows. Section 2 presents background and related work. Section 3 describes the methodology we employed to conduct our work. Then, in Section 4, we report the results for each research question. Section 5 discusses threats to validity. Section 6 concludes the paper and outlines future work. Finally, the Artifact Availability section presents the replication package.

2 Background and Related Work

2.1 Background

Clojure. Clojure is a Lisp dialect [31] designed as a practical functional programming language [15]. It emphasizes immutable data structures, first-class functions, data-oriented programming, and metaprogramming through macros. As a homoiconic language, Clojure represents programs as native data structures, enabling powerful metaprogramming facilities [8]. The language has gained industrial adoption in companies such as Amazon, Nubank, Netflix, Apple, Spotify, Facebook, and Walmart Labs[3].

Many of Clojure's core design decisions were influenced by prior research on persistent data structures, concurrency, and software complexity[4]. For example, Clojure adopts persistent collections based on HAMTs [3] and a Software Transactional Memory model inspired by composable memory transactions [14]. These characteristics influence the structure and maintenance of Clojure systems.

Code Smells. Code smells are recurring structures in source code that suggest design weaknesses and may indicate opportunities for refactoring [12]. Originally proposed by Fowler and Beck in the context of object-oriented software, code smells provide a vocabulary for discussing internal quality problems. While some smells can be observed across different ecosystems, others are closely tied to the constructs, idioms, and development practices of a particular language. In this paper, we use the term *traditional code smells* to refer to smells from Fowler and Beck's original catalog.

Static Analysis in Clojure. The Clojure ecosystem provides several static analysis tools, including *clj-kondo* [6], *Eastwood* [10], *Joker* [17], and *Kibit* [19]. These tools report potential errors, warnings, and style-related issues. Among them, *clj-kondo* is widely adopted and commonly integrated into development environments through *clojure-lsp* [7], enabling editor support in tools such as Visual Studio Code via Calva[5].

2.2 Related Work

Although code smells have been widely studied, few works focus on functional programming languages. Prior work investigated smells and refactoring support in languages such as Haskell [9] and Erlang [4, 11, 21, 22, 25, 27]. However, to the best of our knowledge, no previous study has investigated code smells in Clojure.

The closest work is by Vegi and Valente, who investigated code smells in Elixir through a grey literature review [29]. Their work was later extended into a validated catalog of Elixir-specific smells and refactorings [28, 30]. We follow the same research line, but focus on Clojure, providing early evidence from grey literature, practitioner feedback, and tool support.

3 Methodology

This study adapts the grey literature review conducted by Vegi and Valente for Elixir [29] to Clojure. We preserve the main steps of their early study, including Google search, document selection, data extraction, and the assessment of static analysis support. In addition, we extend the process with an initial interaction with the Clojure community and a practitioner survey to collect feedback on the relevance of the smell candidates. These steps were designed to answer the research questions presented in Section 1.

Figure 1 summarizes our methodology. To answer RQ1 and RQ2, we took the following steps:

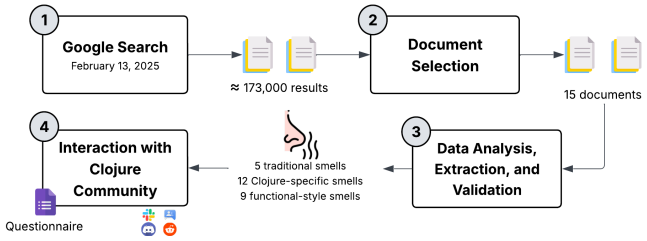

Figure 1: Overview of the research methodology.

1) Google Search: Following the methodology of the Elixir study [29] and guidelines for grey literature reviews [13], we used Google Search to retrieve grey literature documents discussing code smells in Clojure. The search was manually conducted on February 13, 2025, using the query shown in Listing 1, which preserves the original query structure while replacing only the programming language name. The 60 analyzed documents are available in the *Google Top-60.xlsx* spreadsheet included in the replication package [2].

Listing 1: Google search query used in the study

```
("Clojure") AND
("code smell" OR "code smells" OR "bad smell" OR
 "bad smells" OR "anti-pattern" OR "anti-patterns" OR
 "antipattern" OR "antipatterns" OR "bad-practice" OR
 "bad-practices" OR "bad practice" OR "bad practices")
```

[3] https://clojure.org/community/companies
[4] https://clojure.org/about/documentary
[5] https://calva.io/

2) Document Selection: The search returned approximately 173,000 results. Since inspecting all pages would be impractical, grey literature reviews commonly restrict screening to the most relevant results [13]. Following the original study, we manually analyzed Google's top 60 results. The first two authors independently screened the documents and selected those discussing problematic, non-idiomatic, or quality-related practices in Clojure.

After screening, 15 documents were selected for further analysis. The remaining results were excluded because they did not discuss Clojure code quality concerns. No relevant documents were identified among the last ten inspected results, suggesting lower relevance in lower-ranked pages.

For example, the Reddit discussion "Functional programming anti-patterns?" (G1), available in the *Google Top-60.xlsx* spreadsheet of the replication package [2], was selected because practitioners discussed candidate smells such as *Long Method*, *Long Parameter List*, *Deeply-nested Call Stacks*, *Overabstracted Composition*, *Explicit Recursion*, and *Namespaced Keys Neglect*.

3) Data Extraction and Validation: The first two authors independently examined the selected documents to identify excerpts relevant to RQ1 and RQ2. Their assessments were then compared and jointly reviewed to reconcile disagreements and improve consistency. Only two conflicts emerged during this procedure. After discussion, the smell (*Unnecessary Macros*) was retained, whereas the document identified as G8 in the replication package [2] was excluded for not being aligned with the research questions.

4) Interaction with Clojure Community: Between July and September 2025, we publicized the study through Clojure community channels, including Slack, Discord, and Reddit[6]. Discussions in the GitHub repository where the smells were cataloged [1] [7] suggested that traditional smells were perceived as less relevant in the Clojure ecosystem. Community members also suggested conducting a survey to assess the perceived relevance of the identified smells.

Based on this feedback, we conducted a practitioner survey including all candidate smells identified in this study, except traditional smells, which were removed after the initial feedback. For each smell, participants could indicate whether they considered it relevant (*Yes*, *No*, or *I don't know*) and provide additional comments. The survey received 44 responses and 214 optional comments.

In order to answer RQ3, we analyzed the documentation of *clj-kondo*, *Eastwood*, *Joker*, and *Kibit* to determine which traditional and additional smells identified in the Clojure ecosystem are supported by their rules and diagnostics. The identified checks were then mapped to the smells discussed by developers in our dataset.

4 Results

This section presents the results organized by research question. Section 4.1 investigates whether Clojure developers discuss traditional code smells. Section 4.2 analyzes language-specific smells discussed in the Clojure ecosystem. Finally, Section 4.3 examines whether static analysis tools for Clojure detect the reported smells.

[6]https://clojure.org/community/resources

[7]The GitHub catalog is continuously evolving and may not fully reflect the current state of this research.

Table 1: Traditional code smells (grey literature)

Code Smell in Clojure	Documents	#
Long Parameter List	G1, G2	2
Long Method	G1	1
Shotgun Surgery	G14	1
Inappropriate Intimacy	G14	1
Comments	G9	1

4.1 RQ1. Do Clojure Developers Discuss Traditional Code Smells?

Table 1 summarizes the traditional code smells identified in the analyzed documents. Although these smells are discussed by developers, Fowler's terminology is not always explicitly used. Instead, developers often describe these issues through their characteristics.

Long Parameter List was the only smell identified in more than one document (G1 and G2). It was associated with functions containing many parameters and described as difficult to understand and maintain. For example, G2 recommends avoiding functions with many arguments, stating that *"if your functions take a very large number of arguments, suspect they should be broken apart"*.

Other traditional smells also emerged. In G1, *Long Method* appears in recommendations to decompose large functions into smaller composable units, as developers argued that *"it's typically better to write a bunch of small functions"*. In G14, concerns about changing data representations affecting many tests indicate *Shotgun Surgery*, while statements such as *"this test knows too much about the structure of db"* suggest *Inappropriate Intimacy*.

Despite these discussions, community feedback suggested that traditional smells are often perceived as less relevant in Clojure, particularly because many originate from object-oriented languages. This perception was repeatedly expressed in the optional comments provided by participants. For example, participant P_1 noted that:

"The Traditional Smells section lists many code smells that are popular in OOP languages (such as Java). Most of them are not applicable to Clojure..."

Other participants expressed similar concerns, with one suggesting that *"the traditional section can safely be deleted"* (P_2) and P_3 arguing that *"the whole traditional smells section is of dubious use"*.

> **RQ1 Answer:** We found evidence that Clojure developers discuss traditional code smells, although not always using the terminology proposed by Fowler. However, community feedback suggests that these smells are generally perceived as less relevant in the Clojure ecosystem.

4.2 RQ2. Do Clojure Developers Discuss Other Smells?

We organized the identified smells into two categories: (i) *Clojure-specific code smells*, derived from discussions within the Clojure ecosystem, and (ii) *functional-style code smells*, reflecting general characteristics of the functional programming paradigm. The complete catalog is publicly available on GitHub [1] and includes descriptions, examples, evidence sources, and discussion excerpts. The

second category is treated as descriptive evidence observed in the context of Clojure rather than as a formally established taxonomy.

4.2.1 Clojure-Specific Code Smells. Table 2 summarizes the Clojure-specific smells identified in this study, their descriptions, and evidence sources. Overall, these smells reflect non-idiomatic uses of Clojure constructs discussed in community resources and technical documentation. One example is *Unnecessary Macros*, which consists of the inappropriate use of macros in contexts where simple functions would be sufficient. In document G8, it appears as follows:

"Using macros when regular functions would do is a good example of that. It is absolutely possible to write impenetrable Clojure if you start doing weird things just because you can".

As we said before, we collected practitioners' perceptions of the identified Clojure-specific smells (Table 3). Overall, the results show broad agreement with the proposed catalog: 10 out of 12 smells (83.3%) received a majority of positive responses (>50% "Yes"), with half exceeding 70% agreement. These findings suggest that most of the identified smells are recognized by practitioners as problematic coding practices. Conversely, disagreement was limited, as only one smell received a majority of negative responses.

The high level of agreement for *Unnecessary Macros* is also reflected in the optional comments provided by participants. For instance, participant P_4 stated that developers should "avoid macros when you can", relating the smell to the broader principle of choosing the simplest solution for the problem at hand. This comment, along with the remaining survey responses and optional comments, can be found in *Survey Responses.xlsx* in the replication package [2].

Table 3: Perceived relevance of Clojure-specific smells

Code Smell	Yes	No	I don't know	#
Unnecessary Macros	88.37%	6.98%	4.65%	42
Thread Ignorance	76.74%	16.28%	6.98%	43
Nested Forms	76.74%	13.95%	9.30%	43
Redundant do Block	73.81%	16.67%	9.52%	42
Direct Usage of `clojure.lang.RT`	72.09%	9.30%	18.60%	43
Verbose Checks	71.43%	19.05%	9.52%	42
Improper Emptiness Check	60.47%	30.23%	9.30%	43
Conditional Build-up	55.81%	18.60%	25.58%	43
Production `doall`	53.49%	27.91%	18.60%	43
Unnecessary `into`	51.16%	34.88%	13.95%	43
Namespaced Keys Neglect	38.64%	47.73%	13.64%	44
Accessing non-existent Map Fields	26.19%	64.29%	9.52%	42

4.2.2 Code Smells in Functional Style. The second category comprises smells associated with functional programming practices rather than Clojure-specific issues. Table 4 presents the examples identified in the analyzed discussions. These smells are treated as an interpretative synthesis derived from evidence observed in the Clojure context, not as a formally consolidated taxonomy.

An example appears in document G1, available in the *Google Top-60.xlsx* spreadsheet included in the replication package [2], where developers mention practices such as "overabundance of partial application and composition", "designing everything as higher order functions", and "very deep call stacks" as problematic patterns discussed in the context of Clojure and functional programming.

Table 5: Perceived relevance of functional-style smells

Code Smell	Yes	No	I don't know	#
Lazy Side Effects	90.48%	4.76%	4.76%	42
Hidden Side Effects	78.57%	11.90%	9.52%	42
Explicit Recursion	71.43%	23.81%	4.76%	42
Positional Return Values	69.05%	21.43%	9.52%	42
Overabstracted Composition	64.29%	16.67%	19.05%	42
Trivial Lambda	59.52%	30.95%	9.52%	42
Overuse of Higher-order Functions	57.14%	26.19%	16.67%	42
Inefficient Filtering	54.76%	14.29%	30.95%	42
Deeply-nested Call Stacks	48.78%	34.15%	17.07%	41

Among the functional-style smells, practitioner responses show broad agreement with the proposed catalog (Table 5). Overall, 8 out of 9 smells (88.9%) received a majority of positive responses (>50% "Yes"), and none received a majority of negative responses, suggesting general recognition of these practices as potentially problematic. The only smell that did not achieve majority agreement was *Deeply-nested Call Stacks*, while *Inefficient Filtering* exhibited the highest level of uncertainty, indicating that its perceived relevance may depend on context. The full survey responses are provided in *Survey Responses.xlsx*, included in the replication package [2].

RQ2 Answer: We found evidence that Clojure developers discuss additional smells beyond traditional code smells, including both Clojure-specific and functional-style smells. Results from the community interaction and practitioner survey indicate that these smells are perceived as more relevant in the ecosystem than traditional smells.

4.3 RQ3. Do Well-Known Static Code Analysis Tools for Clojure Detect Code Smells?

In our analysis, only 2 of the 26 cataloged smells are explicitly detected by existing tools: *Improper Emptiness Check*, covered by clj-kondo's `:not-empty?`, and *Redundant do Block*, which is covered by `:redundant-do`. A third smell, Verbose Checks, receives partial support via clj-kondo's `:not-nil?`, capturing a common subcase involving explicit nil comparisons. The remaining smells are not directly supported by the analyzed tools. Overall, the results suggest that current Clojure linters focus primarily on local syntactic and idiomatic issues, while most smells remain unsupported.

RQ3 Answer: Existing static analysis tools for Clojure provide only limited support for the cataloged smells. Among the 26 identified smells, only 2 are detected by current tools.

5 Threats to Validity

As a replication study, this work inherits structural threats from the baseline Elixir study [29]. The vulnerabilities are as follows:

Construct Validity. The main risk is missing relevant discussions due to query formulation. To mitigate this, we reused a query from the baseline study, modifying only the language keyword.

Table 2: Clojure-specific code smells, descriptions, and evidence sources

Code Smell	Brief Description	Docs
Unnecessary Macros	Using macros instead of simpler language constructs.	G8
Namespaced Keys Neglect	Using unqualified keywords instead of namespaced ones, increasing ambiguity and the risk of key collisions.	G1
Improper Emptiness Check	Using verbose or non-idiomatic checks to test collection emptiness.	G2
Accessing non-existent Map Fields	Accessing map keys without handling missing values explicitly.	G2
Unnecessary `into`	Using `into` when simpler alternatives exist.	G2
Conditional Build-Up	Building state through multiple conditional updates, leading to verbose, imperative-style code.	G2
Verbose Checks	Reimplementing common checks instead of using built-in predicates.	G2
Production `doall`	Using `doall` in production code to force lazy evaluation, potentially causing memory or performance issues.	G2
Redundant `do` Block	Unnecessary `do` in constructs with implicit sequencing.	G2
Thread Ignorance	Avoiding threading macros in favor of nested forms or bindings.	G2
Nested Forms	Unnecessary nesting of binding/iteration forms instead of combining them.	G2
Direct Usage of `clojure.lang.RT`	Calling internal runtime APIs instead of public Clojure abstractions, leading to fragile and non-portable code.	G5

Table 4: Functional-style code smells, descriptions, and evidence sources

Code Smell	Brief Description	Docs
Trivial Lambda	Overuse of anonymous functions instead of named functions, reducing clarity and reusability.	G1, G2
Inefficient Filtering	Generating invalid values then discarding them via filtering.	G10
Overabstracted Composition	Excessive composition or partial application reducing data-flow readability.	G1
Deeply-nested Call Stacks	Excessive nesting of function calls, making control flow harder to follow and debug.	G1
Overuse of Higher-Order Functions	Excessive use of functions that take or return other functions.	G1
Lazy Side Effects	Side effects executed within lazy evaluation, leading to unpredictable execution behavior.	G12
Hidden Side Effects	Side effects that are not explicit in function structure or naming, reducing predictability and testability.	G1
Explicit Recursion	Use of manual recursion instead of higher-level abstractions such as map, reduce, or filter when applicable.	G1
Positional Return Values	Returning multiple values via positional structures instead of named mappings.	G2

Conclusion Validity. Relying on technical blogs and community channels introduces non-peer-reviewed data that may reflect personal opinions rather than design issues. To reduce subjective bias, the first two authors independently extracted candidate smells and resolved disagreements through joint consensus.

Internal and External Validity. Relying on a single grey literature source may limit coverage of the catalog. We mitigated this by cross-referencing active community hubs to ground findings in practitioner perspectives.

6 Conclusion and Future Work

In this paper, we presented early findings from a replication study investigating code smells in the Clojure ecosystem using a methodology previously applied to Elixir. By reviewing 15 grey literature documents and conducting an initial community consultation, we gathered preliminary evidence regarding how software quality issues manifest in this modern functional language.

For **RQ1**, we found that although traditional code smells are discussed by developers, they are generally perceived as less relevant in Clojure due to the language's functional nature. For **RQ2**, we identified 12 Clojure-specific smells and 9 functional-style code smells, representing recurring non-idiomatic practices in this paradigm. Community interaction and survey responses also allowed us to assess the relevance of these smells. For **RQ3**, we found limited tool support, as Clojure linters focus mainly on syntactic and style issues, leaving most structural and design smells unsupported.

As future work, we plan to expand this study in three directions. First, we intend to mine open-source GitHub Clojure repositories, analyzing source code, commits, issues, and pull requests to identify additional smells and gather further empirical evidence. Second, we are developing ARIT, a static analysis tool that currently detects 20 Clojure-specific smells from our catalog [20] and is being extended to cover a broader range of them. In parallel, we are contributing new *clj-kondo* rules based on the catalog presented in this work [1]. Finally, we plan to investigate refactoring practices and propose a catalog of strategies for removing both Clojure-specific and functional-style smells.

Artifact Availability

The replication package of this study includes: (i) the complete catalog of identified code smells (*Code Smells Catalog.xlsx*); (ii) the survey questionnaire (*Survey.pdf*); (iii) anonymized survey responses (*Survey Responses.xlsx*); (iv) the quantitative survey analysis (*Survey Analysis.xlsx*); and (v) the list of analyzed Google results (*Google Top-60.xlsx*). The package is publicly available online [2].

Acknowledgements

We thank the Clojure community members for participating in this study and providing valuable feedback that helped refine and enrich our code smells catalog. This work was developed under the NuFuturo project, a collaboration between the Federal University of Campina Grande and Nubank.

References

[1] Walber Araújo, José Truta, Rafael Serey, Thiago Laurentino, and João Brunet. 2025. clj-smells-catalog: Catalog of Clojure-related code smells. https://github.com/nufuturo-ufcg/clj-smells-catalog. GitHub repository. Accessed: May 20, 2026.

[2] Walber Araújo, José Truta, Lucas Vegi, Marco Tulio Valente, and João Brunet. 2026. *Replication Package: Code Smells in the Clojure Ecosystem.* doi:10.5281/zenodo.20293947

[3] Phil Bagwell. 2001. *Ideal Hash Trees.* Technical Report. EPFL. Available at: https://infoscience.epfl.ch/server/api/core/bitstreams/f66a3023-2cd0-4b26-af6e-91a9a6ae7450/content.

[4] Christopher M. Brown and Simon Thompson. 2010. Clone Detection and Elimination for Haskell. In *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM).* 111–120. doi:10.1145/1706356.1706378

[5] Suelen Goularte Carvalho, Maurício Aniche, Júlio Veríssimo, Rafael S. Durelli, and Marco Aurélio Gerosa. 2019. An empirical catalog of code smells for the presentation layer of Android apps. *Empirical Software Engineering* 24 (2019), 3546–3586. doi:10.1007/s10664-019-09768-9

[6] clj-kondo Contributors. 2026. clj-kondo. https://github.com/clj-kondo/clj-kondo. Accessed: 2026-05-14.

[7] clojure-lsp contributors. 2024. Clojure LSP. https://clojure-lsp.io/. Accessed: 2026-05-14.

[8] Clojure.org. 2026. The Reader. https://clojure.org/reference/reader Accessed: 2026-05-18.

[9] Jonathan Cowie. 2005. *Detecting bad smells in Haskell.* Technical Report. University of Kent, UK.

[10] Eastwood Contributors. 2024. Eastwood: Clojure Linter. https://github.com/jonase/eastwood. Accessed: 2026-05-14.

[11] Viktória Fördős and Melinda Tóth. 2016. Identifying Code Clones with RefactorErl. *Acta Cybernetica* 22, 3 (2016), 553–571. doi:10.14232/actacyb.22.3.2016.1

[12] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. 1999. *Refactoring: Improving the Design of Existing Code* (1st ed.). Addison-Wesley, Boston, MA, USA.

[13] Vahid Garousi, Michael Felderer, and Mika V. Mäntylä. 2019. Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. *Information and Software Technology* 106 (2019), 101–121. doi:10.1016/j.infsof.2018.09.006

[14] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. 2005. Composable memory transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Chicago, IL, USA) *(PPoPP '05).* Association for Computing Machinery, New York, NY, USA, 48–60. doi:10.1145/1065944.1065952

[15] Rich Hickey. 2008. The Clojure programming language. In *Proceedings of the 2008 Symposium on Dynamic Languages* (Paphos, Cyprus) *(DLS '08).* Association for Computing Machinery, New York, NY, USA, Article 1, 1 pages. doi:10.1145/1408681.1408682

[16] Rich Hickey. 2026. Clojure Programming Language. https://clojure.org/. Accessed: 2026-05-14.

[17] joker Contributors. 2026. Joker: Small Clojure Interpreter, Linter, and Formatter. https://github.com/candid82/joker. Accessed: 2026-05-19.

[18] Frederick P. Brooks Jr. 1982. *The Mythical Man-Month: Essays on Software Engineering.* Addison-Wesley, Reading, Massachusetts.

[19] Kibit Contributors. 2024. Kibit. https://github.com/clj-commons/kibit. Accessed: 2026-05-14.

[20] Thiago Laurentino, Rafael Serey, Walber Araújo, Natália Rodrigues, and João Brunet. 2026. ARIT: A Tool for Detecting Code Smells in Clojure Programs. In *Proceedings of the XVII Brazilian Conference on Software: Theory and Practice (CBSoft), Tool Session.* Sociedade Brasileira de Computação (SBC), São Paulo, Brazil. To appear.

[21] Huiqing Li and Simon Thompson. 2009. Clone Detection and Removal for Erlang/OTP within a Refactoring Environment. In *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM).* 169–178. doi:10.1145/1480945.1480971

[22] Huiqing Li and Simon Thompson. 2010. Refactoring Support for Modularity Maintenance in Erlang. In *10th IEEE Working Conference on Source Code Analysis and Manipulation (SCAM).* 157–166. doi:10.1109/SCAM.2010.17

[23] T. Mens and T. Tourwe. 2004. A survey of software refactoring. *IEEE Transactions on Software Engineering* 30, 2 (2004), 126–139. doi:10.1109/TSE.2004.1265817

[24] PMD Developers. 2024. PMD: Source Code Analyzer for Java, JavaScript, Salesforce, and More. https://pmd.github.io. Accessed: 2026-05-14.

[25] Pablo Lamela Seijas and Simon Thompson. 2016. Identifying and Introducing Interfaces and Callbacks Using Wrangler. In *28th Symposium on the Implementation and Application of Functional Programming Languages (IFL).* 1–13. doi:10.1145/3064899.3064909

[26] SonarSource. 2024. SonarQube: Continuous Inspection of Code Quality. https://www.sonarqube.org. Accessed: 2026-05-14.

[27] Simon Thompson, Huiqing Li, and Andreas Schumacher. 2017. The pragmatics of clone detection and elimination. *The Art, Science, and Engineering of Programming* 1, 2 (2017), 1–34.

[28] Lucas Vegi and Marco Valente. 2025. Code Smells and Refactorings for Elixir. In *Anais do XXXVIII Concurso de Teses e Dissertações* (Maceió/AL). SBC, Porto Alegre, RS, Brasil, 94–103. doi:10.5753/ctd.2025.9280

[29] Lucas Francisco da Matta Vegi and Marco Tulio Valente. 2022. Code smells in Elixir: early results from a grey literature review. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension (ICPC '22).* ACM, 580–584. doi:10.1145/3524610.3527881

[30] Lucas Francisco Matta Vegi and Marco Tulio Valente. 2023. Understanding code smells in Elixir functional language. *Empirical Software Engineering* 28, 102 (2023), 1–32. doi:10.1007/s10664-023-10343-6

[31] Patrick Henry Winston and Berthold K Horn. 1986. Lisp. (1986).

[32] Haiyin Zhang, Luís Cruz, and Arie Van Deursen. 2022. Code Smells for Machine Learning Applications. In *Proceedings - 1st International Conference on AI Engineering - Software Engineering for AI, CAIN 2022.* IEEE/ACM, Pittsburgh, PA, USA, 217–228. doi:10.1145/3522664.3528620