Detecting Code Smells in Elixir Using Metrics and Predicates for Functional Programming

Erick Soares de Souza
Department of Computer Science, Federal University of Minas Gerais (UFMG)
Belo Horizonte, Brazil
ericksoares@dcc.ufmg.br

João Marcelo F. de Almeida
Department of Computer Science, Federal University of Minas Gerais (UFMG)
Belo Horizonte, Brazil
joao.freitas.almeida@dcc.ufmg.br

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

Mateus Dias
Department of Informatics, Federal University of Viçosa (UFV)
Viçosa, Brazil
mateus.j.dias@ufv.br

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

Abstract

Code smells are design problems that hinder long-term code maintenance. Although widely studied in object-oriented languages, functional languages such as Elixir still lack mechanisms for the automatic detection of these problems. Although previous work cataloged 35 code smells specific to Elixir, no metrics or tools have been proposed for their automatic detection. This paper proposes metrics and predicates for detecting 8 of the 9 *Low-Level Concerns* in this catalog, a category composed of smells that affect small and isolated code structures. The validation is performed on ExSmell-Train, a dataset containing examples of code smells in Elixir, achieving recall equal to or greater than 80% in all analyzed cases.

Keywords

Code Smell, Elixir, Functional Programming, Software Metrics, Static Analysis

1 Introduction

Code smells are design problems in code that do not prevent compilation nor cause immediate errors, but hinder the long-term maintenance of the system [12]. In the literature, these problems are widely studied in object-oriented languages, where metrics such as those of Chidamber and Kemerer (CK) [5], defined in 1994, are used for the automatic detection of smells in code.

In contrast, functional languages still lack comprehensive studies on the automatic detection of code smells [24]. One example is the Elixir language [20], which runs on Erlang's BEAM virtual machine, granting it native support for high concurrency and fault tolerance. Because of these characteristics, it is widely adopted by companies and large-scale systems, such as Discord [6], Bleacher Report [7], and PepsiCo [22]. Recently, Vegi *et al.* cataloged and proposed refactorings for 35 code smells in Elixir [23], of which 23 are language-specific and are divided into the *Design-Related* and *Low-Level Concerns* categories.

Even with this cataloging, a gap remains in the automatic detection of these problems. Thus, this paper proposes metrics and predicates to identify code smells in Elixir belonging to the *Low-Level Concerns* group, since they affect small, isolated code structures that are better suited for detection based on local metrics. The validation is performed on ExSmell-Train[1], a dataset containing 10 positive examples for each smell.

The remainder of this paper is organized as follows: Section 2 presents the theoretical background and discusses related work. Section 3 details the proposed metrics and predicates. Section 4 describes the evaluation performed. Finally, Section 5 presents the conclusions and future work.

2 Background and Related Work

2.1 Code Smells and Software Metrics

Fowler defines code smells as surface indications of deeper design or structural problems in code, having cataloged 22 smells [12]. For the automatic detection of these problems in object-oriented languages, the most widely used metrics are those of Chidamber and Kemerer (CK) [5]. Among the most common metrics are Lines of Code (LOC) [9], which counts the number of lines of code in a function or module, and Cyclomatic Complexity (CC) [16], which measures how many independent logical paths exist in the code.

Although several consolidated metrics exist for detecting *code smells* in object-oriented systems, it remains unclear whether they are suitable for functional languages. Programming language characteristics native to this paradigm, such as immutability and the absence of shared state, mitigate several classic object-oriented design problems while introducing different challenges [24]. Consequently, existing object-oriented metrics may not capture these functional-specific issues, highlighting the need to investigate and develop approaches tailored to the functional context [23, 24].

2.2 The Elixir Language

Elixir is a functional language created in 2011 by José Valim at the company Plataformatec [20]. The language runs on the BEAM virtual machine, inherited from Erlang, and was designed to be general-purpose and highly concurrent [3]. Its main characteristics include immutability, pattern matching, and lightweight processes, in addition to a more accessible learning curve compared to other functional languages [13].

[1] https://github.com/labd2m/ExSmell-Train

Table 1: Low-Level Concerns cataloged by Vegi *et al.* [24]

Smell	Description
Accessing Non-Existent Map/Struct Fields	Dynamic access to fixed fields hides errors
Alternative Return Types	Inconsistent return types make behavior unpredictable
Complex Branching	Centralized handling of multiple errors hinders comprehension
Complex else Clauses in With	Grouping all errors into the else block hides the exact origin of the failure
Dynamic Atom Creation	Dynamic atom creation can exhaust the BEAM's global table
Modules with Identical Names	Duplicate modules overwrite each other in the BEAM
Speculative Assumptions	Speculative assumptions about the code's behavior
Unnecessary Macros	Use of macros where functions would suffice
Working with Invalid Data	Absence of validation propagates invalid data

2.3 Elixir Code Smells

Vegi *et al.* [24] cataloged 35 code smells specific to Elixir, divided into two categories: *Design-Related*, composed of 14 smells that affect the macro organization of the system, and *Low-Level Concerns*, composed of nine smells that affect small and isolated code structures, such as functions and modules [23]. Table 1 lists the nine smells in the *Low-Level Concerns* category.

2.4 Related Work

The automatic detection of code smells in object-oriented languages has been extensively studied [10]. According to Sobrinho et al. [19], at least 351 studies on code smells were published between 1990 and 2017, with smell detection being the most common research topic, accounting for approximately 30% of the studies. This body of work has led to the proposal of around 80 detection techniques and tools, many of which have been comparatively evaluated [10]. Representative examples include PMD [18], which detects smells through static rules over the Abstract Syntax Tree (AST); JDeodorant [21], which relies on cohesion and coupling metrics to identify refactoring opportunities; and Checkstyle [4], which focuses on style patterns and complexity. The detection strategies employed by these tools are largely grounded on metric-based approaches proposed in the literature, such as those by Marinescu [14, 15], who introduced metrics and thresholds for detecting smells including God Class and Data Class. However, these tools and metrics were designed for object-oriented languages and rely on concepts such as classes, inheritance, and methods, which do not directly apply to the functional paradigm.

In functional languages, studies remain scarce [24]. For Elixir specifically, despite the existence of Vegi *et al.*'s catalog [23], no metrics or tools have been proposed for the automatic detection of the identified smells. Credo [11], the leading static analysis tool in the Elixir community, offers style and general best-practice checks, but does not cover the specific smells in Vegi *et al.*'s catalog.

Furthermore, consolidated methods for deriving software metric thresholds, such as that of Erni & Lewerentz [8], rely on statistical assumptions (such as approximately unimodal distributions) that do not always hold for real-world code size and complexity metrics, which tend to exhibit strong right-skewness. Alves *et al.* [1] propose an alternative approach, based on percentiles calculated over a benchmark of real projects, which is more robust to this type of distribution. Similarly, Oliveira *et al.* [17] introduced the concept of relative thresholds to evaluate source code metrics by analyzing a benchmark of systems, further reinforcing the need for empirically grounded thresholds.

3 Metrics and Predicates

This section presents the metrics and predicates proposed for the automatic detection of the *Low-Level Concerns* in Vegi *et al.*'s catalog [24]. Speculative Assumptions is the only smell in this group not addressed in this work, as its detection requires deeper semantic understanding of the code's runtime behavior, which suggests a possibly hybrid approach combining metrics, predicates and language models.

3.1 Metrics Used

For the detection of the *smells*, we use metrics already consolidated in the object-oriented literature, adapted to Elixir's functional context, in addition to a new metric proposed by this work. The metrics used are:

- **Lines of Code (LOC):** counts the number of lines of code in a function.
- **Cyclomatic Complexity (CC):** measures cyclomatic complexity, counting the independent logical paths in the function.
- **Clause Else Count (CEC):** a metric proposed in this work that counts the total number of clauses present specifically within the `else` block of a `with` construct. Since the metric is evaluated only on expressions that have this block, there are no null values (CEC is always ≥ 1).

As a practical example of the CEC metric, consider the structure in Figure 1. In this case, the function receives a score of CEC = 3, since there are three handling clauses (lines 5–7) defined within the `else` block of the `with` structure:

3.2 Threshold Definition

To compute the Complex Branching and Complex else Clauses in With metrics, the thresholds were derived from a real-code benchmark using the automatic distribution-cropping method proposed by Fontana *et al.* [2]. This benchmark comprises 20 popular open-source Elixir repositories hosted on GitHub, including notable projects such as Phoenix, Ecto, and Oban, from which the LOC,

```
with {:ok, user} <- fetch_user(id),
     {:ok, profile} <- fetch_profile(user) do
  render(user, profile)
else
  {:error, :not_found} -> handle_not_found()
  {:error, :unauthorized} -> handle_unauthorized()
  _ -> handle_generic_error()
end
```

Figure 1: Example of the CEC metric (CEC = 3).

CC, and CEC values of all functions were extracted. The extraction was performed by implementing a custom static analysis rule (referred to as a *check*) in Credo [11], which traverses the AST of the analyzed projects to compute these metrics for each function declaration. The exact threshold values were then computed from the extracted data using a dedicated script implementing Fontana *et al.*'s method [2]. The list of repositories, the implementation of the custom checks, the extracted metric values, and the threshold computation script are all available in our replication package[2].

This distribution-cropping method identifies a cut point v_{mid} that separates the dense, low-variability portion of the metric distribution from its long, more variable tail. This separation is necessary because the dense portion is typical of the many small, simple functions found in a codebase, whereas potentially smelly functions are more likely to reside in the long tail. To locate this cut point, the metric's quantile function is sampled at 100 equally spaced points. The frequency with which each resulting value repeats among these samples is then computed, and v_{mid} is defined as the smallest such value from which point onward every sampled frequency falls at or below their overall median. Values below v_{mid} are discarded, and the 75th percentile (P_{75}) of the remaining, cropped distribution is taken as the final threshold. Table 2 presents the thresholds obtained for the numeric metrics used in the detection of the *smells*.

Table 2: Thresholds calculated over the repositories

Range	LOC	CC	CEC
P_{75} of values $\geq v_{mid}$	33	8	5

Figure 2 illustrates this quantile function and the calculated cut point for the LOC metric. As shown, the curve grows very slowly for most benchmark functions, remaining nearly flat until approximately the 89th percentile. At this coordinate, the algorithm identifies $v_{mid} = 15.0$ as the cut point from which the distribution is cropped, and the 75th percentile of the remaining values yields the final threshold. Everything to the left of this line represents the dense, repetitive core of small operations discarded from the calculation, while the steep rise to the right captures the long tail of complex functions.

Figure 3 details the frequency-based criterion used to select v_{mid}. It shows the same LOC values sampled from the quantile function, sorted in descending order along the y-axis, where bar width reflects how many times each value repeats among the 100 sampled points (x-axis). The overall median of these frequencies, f_{mid}, is computed, and v_{mid} is defined as the smallest sampled value from which point onward every remaining frequency falls at or below f_{mid}, as illustrated by the widening bars toward the top.

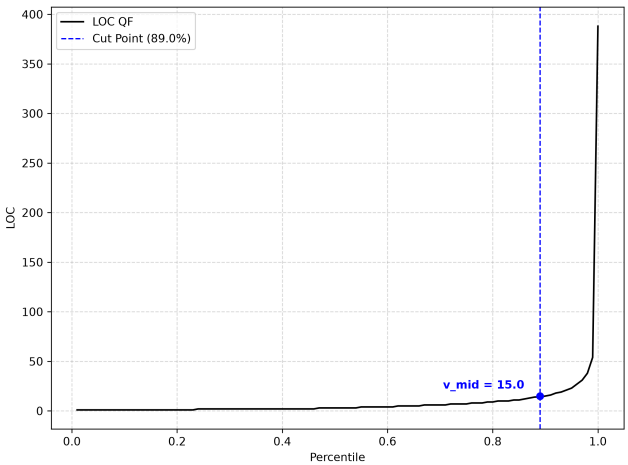

Figure 2: LOC percentile curve with cut point.

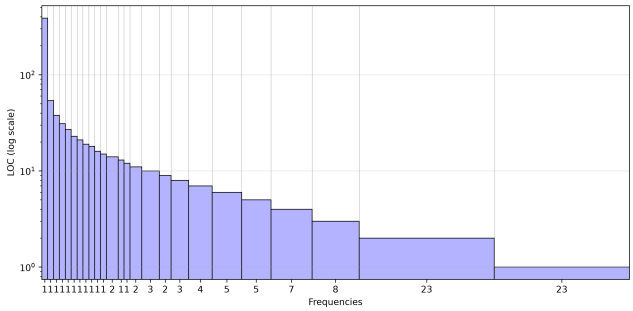

Figure 3: Frequency of each unique LOC value among the sampled quantiles, used to derive v_{mid}.

3.3 Metrics and Predicates

In addition to the numeric metrics, we use binary predicates for *smells* that involve static structural checks. From a technical standpoint, the implementation and evaluation of these predicates on Elixir code is made possible by the language's metaprogramming features. Elixir source code can be natively converted into its AST, where expressions are represented by three-element tuples. Thus, the proposed predicates can act as *pattern matching* rules that scan the AST in search of specific structural patterns, ensuring a purely static analysis that dispenses with the need for code execution. Table 3 summarizes the proposed detection rules for each *smell*, which are then discussed in detail in the following subsections.

[2]https://github.com/ericksoares13/low_level_concerns

Table 3: Metrics and predicates proposed by smell

Code Smell	Metric/Predicate
Accessing Non-Existent Map/Struct Fields	$DAA = \top$
Alternative Return Types	$ART = \top$
Complex Branching	$LOC \geq 33 \wedge CC \geq 8$
Complex *else* Clauses in with	$CEC \geq 5$
Dynamic Atom Creation	$DAC = \top$
Modules with Identical Names	$1 \leq NCL \leq 2$
Unnecessary Macros	$UM = \top$
Working with Invalid Data	$WID = \top$

3.3.1 *Accessing Non-Existent Map/Struct Fields.*

This smell is detected by the Dynamic Access Analysis predicate (*DAA*), which checks for the use of the dynamic access syntax `struct[:atom_literal]` in functions where the structure has fixed fields. Variables conventionally considered dynamic, such as `params`, `opts`, and `options`, are excluded from the analysis. This exclusion list is parameterizable, based on Credo's custom *checks*.

3.3.2 *Alternative Return Types.*

The Alternative Return Types predicate (*ART*) is true if the function receives configuration parameters (such as `opts`, `options`, `flags`) and contains a `case` structure whose evaluated expression is `opts[:atom_literal]`. The list of variables recognized as dynamic is also parameterizable.

3.3.3 *Complex Branching.*

This smell is detected by the logical expression $LOC \geq 33 \wedge CC \geq 8$. The thresholds were derived from the benchmark of real repositories using the distribution-cropping method described in Section 3.2. The combination of these two metrics accurately identifies long functions that have an excessive number of branches and high structural complexity, capturing the essence of the problem.

3.3.4 *Complex else Clauses in with.*

The smell occurs when $CEC \geq 5$. The threshold was derived using the same distribution-cropping method described in Section 3.2. Grouping multiple clauses into a single `else` block of a `with` construct merges error handling from different origins, hindering the traceability of the failure.

3.3.5 *Dynamic Atom Creation.*

The Dynamic Atom Creation predicate (*DAC*) identifies the creation of atoms at runtime, typically via `String.to_atom/1`, from user input or responses from external APIs. Since atoms are never collected by the Garbage Collector and the BEAM has a global limit of approximately one million entries, unrestricted dynamic creation can exhaust the atom table and crash the virtual machine [13].

3.3.6 *Modules with Identical Names.*

Detected when $1 \leq NCL \leq 2$, where *Naming Conflict Level* (*NCL*) represents the level of naming conflict. If $NCL = 0$, the module reflects the file path with a full *namespace* (no *smell*). If $NCL = 1$, it has a *namespace*, but does not reflect the directory structure (potential risk). If $NCL = 2$, it has no *namespace* or its name is identical to another module in the project (confirmed *smell*). Since the BEAM can only load one instance of each module at a time, the second module overwrites the first in memory [24].

3.3.7 *Unnecessary Macros.*

The Unnecessary Macros predicate (*UM*) is true when a macro's body contains only a simple `unquote` of its arguments, without performing any transformation on the values. In this case, a regular function would be sufficient to express the same behavior. Macros are powerful metaprogramming mechanisms, but their unnecessary use makes code more complex, less readable, and harder to maintain.

3.3.8 *Working with Invalid Data.*

The Working with Invalid Data predicate (*WID*) identifies functions that do not validate the types of their parameters before use. The absence of validation can propagate invalid data to internal or third-party libraries, generating errors at points far removed from the original code and hindering the identification of the problem.

4 Evaluation

4.1 Dataset

The validation of the proposed metrics and predicates was performed on ExSmell-Train, a dataset composed of 350 examples of code smells in Elixir, with 10 examples for each of the 35 smells cataloged by Vegi *et al.* [24]. The examples were generated by the ChatGPT-5.4 model and manually validated by the dataset's authors. For this work, we used the examples corresponding to the eight *Low-Level Concerns* addressed in this paper (Sections 1 and 3), a category chosen for its focus on small, isolated code structures amenable to local metric-based detection, totaling 80 positive examples.

It is worth noting that, unlike the other metrics and predicates, the Complex Branching thresholds were not calibrated on ExSmell-Train itself, but rather on an independent benchmark of real repositories (Section 3.2), which avoids the threat of calibrating and evaluating on the same dataset.

Since ExSmell-Train contains only positive examples, the evaluation was conducted with a focus on recall, which measures the proportion of correctly identified positive examples. The absence of negative examples prevents the calculation of precision and F1-score, as discussed in Section 4.4.

4.2 Results

Table 4 presents the recall obtained for each *smell* evaluated. The approach achieved 100% recall for six of the eight smells analyzed; two cases fell below 100%: Accessing Non-Existent Map/Struct Fields (90%) and Complex Branching (80%).

In the case of Accessing Non-Existent Map/Struct Fields, the undetected example (example 2) used the variable `params`, which is on the exclusion list for being conventionally dynamic in web contexts. However, the example illustrates a case in which dynamic access is in fact problematic, since `params[:discount_code]` treats a missing key and a null value identically, introducing ambiguity, as illustrated in Figure 4.

This highlights a limitation of the variable-name-based exclusion heuristic, which may be overly conservative in certain contexts, and suggests that future refinements could incorporate type or context information to reduce false negatives.

Table 4: Recall obtained by smell on ExSmell-Train

Code Smell	Recall
Accessing Non-Existent Map/Struct Fields	90%
Alternative Return Types	100%
Complex Branching	80%
Complex *else* Clauses in with	100%
Dynamic Atom Creation	100%
Modules with Identical Names	100%
Unnecessary Macros	100%
Working with Invalid Data	100%

```
def create(params) do
  if params[:discount_code] do
    {:ok, %{mode: :discounted,
            code: params[:discount_code]}}
  else
    {:ok, %{mode: :regular}}
  end
end
```

Figure 4: Example of the Accessing Non-Existent Map/Struct Fields smell.

In the case of Complex Branching, the two undetected examples (examples 5 and 9) presented LOC values of 29 and 31, respectively, both below the derived threshold of 33, despite exhibiting high cyclomatic complexity (CC = 13 and CC = 15). This suggests that these examples represent structurally complex but relatively compact functions, a pattern that the LOC threshold alone fails to capture.

4.3 Preliminary Precision Results

As a preliminary check on precision, we additionally applied the Complex Branching check to ExSmell-Gold[3], a companion dataset of 3,500 Elixir code examples containing both smelly and smell-free instances, from which we used the approximately 1,700 functions manually validated as smell-free. Of these, only one was flagged as a false positive, indicating that the derived threshold rarely misclassifies code that does not exhibit this smell.

4.4 Threats to Validity

Internal validity. The thresholds for the numeric metrics were calculated on the benchmark of 20 real GitHub repositories. Using a larger number of repositories for calibration could produce even more representative and robust thresholds.

External validity. ExSmell-Train was generated by a language model (ChatGPT-5.4) and manually validated by sampling. Examples generated by LLMs may not faithfully reflect code patterns found in real production projects, which may limit the generalization of the results. Nonetheless, ExSmell-Train still provides a controlled and reproducible benchmark for evaluating recall: the examples were manually reviewed by the dataset's authors to ensure conformance with the smell definitions, and the use of a consistent generation process facilitates replication.

Construct validity. The ExSmell-Train dataset contains only positive instances of *code smells* and does not include negative examples (*i.e.*, clean code). Consequently, precision and F1-score cannot be computed, preventing a comprehensive evaluation of the proposed approach. To partially mitigate this limitation, we conducted a preliminary precision assessment using the ExSmell-Gold dataset, which includes both smell instances and clean code examples. Furthermore, some predicates are known to produce false positives. For example, the Unnecessary Macros predicate relies on the presence of `unquote`, a heuristic that may not capture all contexts in which a macro is actually unnecessary.

5 Conclusion and Future Work

This paper proposed metrics and predicates for the automatic detection of eight of the nine *Low-Level Concerns* in Vegi *et al.*'s catalog [24] for the Elixir language. Metrics adapted from the object-oriented literature (LOC and CC) were combined with a new metric (CEC) and six binary predicates specific to the language's functional context. Thresholds were derived using the automatic distribution-cropping method of Fontana *et al.* [2] applied to a benchmark of 20 real-world Elixir repositories. The validation on ExSmell-Train achieved recall of 80% in all analyzed cases, demonstrating the feasibility of the approach

Among the main limitations of this work is the fact that ExSmell-Train contains only positive examples, which makes it impossible to evaluate precision and F1-score. In addition, some predicates have known false positives, such as Unnecessary Macros, whose detection criterion based on simple `unquote` may not capture all contexts in which a macro is indeed unnecessary.

As future work, we intend to: (i) implement the proposed metrics and predicates as custom *checks* in Credo, a static analysis tool widely used in the Elixir community; (ii) evaluate the approach on ExSmell-Gold, a dataset with 3,500 examples of Elixir code generated by Claude Sonnet 4.6 containing both positive and negative examples, which will allow the calculation of precision and F1-score in addition to recall; (iii) extend the approach to the *Design-Related Concerns*, a category with 14 smells that affect the macro organization of the system and that still lacks automatic detection mechanisms; and (iv) demonstrate the applicability of the metrics through the automated integration of these checks into CI/CD pipelines, preventing these components from being introduced into the production repository.

Artifact Availability

The replication package of our study is available at: https://github.com/ericksoares13/low_level_concerns.

Acknowledgments

This research was supported by grants from FAPEMIG (grant APQ-02419-23) and CNPq (grant 403304/2025-3).

[3]https://github.com/labd2m/ExSmell-Gold

References

[1] Tiago L. Alves, Christiaan Ypma, and Joost Visser. 2010. Deriving Metric Thresholds from Benchmark Data. In *26th IEEE International Conference on Software Maintenance (ICSM)*. 1–10. doi:10.1109/ICSM.2010.5609747

[2] Francesca Arcelli Fontana, Vincenzo Ferme, Marco Zanoni, and Aiko Yamashita. 2015. Automatic Metric Thresholds Derivation for Code Smell Detection. In *6th IEEE/ACM International Workshop on Emerging Trends in Software Metrics (WETSoM)*. 44–53. doi:10.1109/WETSoM.2015.14

[3] Joe Armstrong. 2007. A history of Erlang. In *3rd ACM SIGPLAN conference on History of programming languages (HOPL)*. 6–1.

[4] Checkstyle Contributors. 2026. Checkstyle. https://checkstyle.sourceforge.io/.

[5] Shyam R Chidamber and Chris F Kemerer. 1994. A metrics suite for object oriented design. *IEEE Transactions on software engineering* 20, 6 (1994), 476–493.

[6] Elixir Team. 2020. Real Time Communication at Scale with Elixir at Discord. https://elixir-lang.org/blog/2020/10/08/real-time-communication-at-scale-with-elixir-at-discord/ Accessed: 2026.

[7] Erlang Solutions. 2021. Bleacher Report Case Study. https://www.erlang-solutions.com/case-studies/bleacher-report-case-study/ Accessed: 2026.

[8] Klaus Erni and Claus Lewerentz. 1996. Applying design-metrics to object-oriented frameworks. In *3rd International Software Metrics Symposium (METRICS)*. 64–74.

[9] Norman Fenton and James Bieman. 2014. *Software metrics: a rigorous and practical approach*. CRC press.

[10] Eduardo Fernandes, Johnatan Oliveira, Gustavo Vale, Thanis Paiva, and Eduardo Figueiredo. 2016. A review-based comparative study of bad smell detection tools. *Information and Software Technology* 74 (2016), 36–49.

[11] René Föhring. 2026. Credo: A static code analysis tool for the Elixir language. https://github.com/rrrene/credo.

[12] Martin Fowler. 2018. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

[13] Saša Jurić. 2024. *Elixir in action* (3 ed.). Manning.

[14] R. Marinescu. 2001. Detecting design flaws via metrics in object-oriented systems. In *39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS)*. 173–182.

[15] R. Marinescu. 2005. Measurement and quality in object-oriented design. In *21st IEEE International Conference on Software Maintenance (ICSM)*. 701–704.

[16] Thomas J McCabe. 1976. A complexity measure. *IEEE Transactions on software Engineering* 4 (1976), 308–320.

[17] Paloma Oliveira, Marco Tulio Valente, and Fernando Lima. 2014. Extracting Relative Thresholds for Source Code Metrics. In *IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*. 254–263.

[18] PMD Contributors. 2026. PMD Source Code Analyzer. https://pmd.github.io/.

[19] Elder Sobrinho, Andrea De Lucia, and Marcelo Maia. 2021. A systematic literature review on bad smells–5 w's: which, when, what, who, where. *IEEE Transactions on Software Engineering* 47, 1 (2021), 17–66. doi:10.1109/TSE.2018.2880977

[20] The Elixir Team. [n. d.]. Elixir Programming Language. https://elixir-lang.org

[21] Nikolaos Tsantalis and Alexander Chatzigeorgiou. 2008. JDeodorant: Identification and application of extract class refactorings. In *12th European Conference on Software Maintenance and Reengineering (CSMR)*. 275–279.

[22] José Valim. 2021. Marketing and Sales Intelligence with Elixir at PepsiCo. https://elixir-lang.org/blog/2021/04/02/marketing-and-sales-intelligence-with-elixir-at-pepsico/ Accessed: 2026.

[23] Lucas Francisco da Matta Vegi. 2024. *Code Smells and refactorings for Elixir*. Ph.D. Thesis. Universidade Federal de Minas Gerais.

[24] Lucas Francisco da Matta Vegi and Marco Tulio Valente. 2023. Understanding code smells in Elixir functional language. *Empirical Software Engineering* 28, 4 (2023), 1–36.