Code Smells and Refactorings for Elixir

Lucas Francisco da Matta Vegi[1] **and Marco Tulio Valente (Advisor)**[1]

[1]Department of Computer Science - Federal University of Minas Gerais (UFMG)
Belo Horizonte – MG – Brazil

{lucasvegi,mtov}@dcc.ufmg.br

Abstract. *Elixir is a modern functional language gaining traction in the industry, but the internal quality of the code written with this language still lacks research. To fill this gap, this thesis investigates code smells and refactorings specific to Elixir, drawing inspiration from classic Fowler catalogs. In the first two studies, a mixed-method approach was adopted to catalog 35 code smells (23 novel and 12 traditional), validated by 181 developers, and 82 refactorings (14 novel), validated by 151 developers. In the third study, we correlated code smells and refactorings, establishing practical guidelines for the disciplined removal of smells through refactorings. The results impact both the prevention of code smells and the prioritization of refactorings in Elixir, for instance.*

Resumo. *Elixir é uma linguagem funcional moderna em ascensão na indústria, mas a qualidade interna dos códigos criados com essa linguagem ainda carece de estudos. Para preencher essa lacuna, esta tese investiga code smells e refatorações específicos para Elixir, inspirando-se nos catálogos clássicos de Fowler. Nos dois primeiros estudos, uma abordagem metodológica mista foi adotada para catalogar 35 code smells (23 inéditos e 12 tradicionais), validados por 181 desenvolvedores, e 82 refatorações (14 inéditas), validadas por 151 desenvolvedores. No terceiro, correlacionamos smells e refatorações, estabelecendo diretrizes para a remoção disciplinada dos smells. Os resultados têm implicações na prevenção de smells e na priorização de refatorações em Elixir.*

1. Introduction

The concern for ensuring product quality is a common characteristic among engineering disciplines, and software engineering is no exception. The assessment of software quality can be classified into two categories: *external quality* and *internal quality* [Meyer 1997]. Based on this definition, the external quality of software measures quality attributes that do not depend on source code to be evaluated, such as robustness, correctness, usability, and efficiency. On the other hand, the internal quality of software evaluates quality attributes directly related to its code, such as maintainability, testability, and readability.

Successive maintenance activities gradually increase the complexity and difficulty of maintaining a system's code and internal structure. Essentially, these maintenance interventions contribute to the progressive degradation of the system's internal quality over time [Lehman 1980]. This fact is also mentioned by Fowler in the preface of his well-known book on refactoring [Fowler and Beck 1999], where he recounts an experience as a consultant on a project that failed for this reason: *"...the project failed, in large part because the code [became] too complex to debug or to tune to acceptable performance"*.

This same failed project was successfully restarted and maintained by applying refactoring techniques, which are code transformations aimed at improving the quality of a system without altering its behavior [Fowler and Beck 1999], thereby stabilizing the natural decline in quality of these systems after successive maintenance activities.

Fowler's success with this project inspired his book on refactoring [Fowler and Beck 1999], where he cataloged 72 object-oriented refactorings, popularizing these techniques. Additionally, Fowler and Beck coined the term *"code smell"* for sub-optimal code structures that hinder software evolution, identifying 22 such smells as refactoring opportunities.

Since the impacts on software quality caused by code smells are not homogeneous and can vary across different domains [Fontana et al. 2013], developers' perception of code smells can also differ [Taibi et al. 2017]. Particularly, each programming language has its own constraints and challenges to perform refactorings [Li and Thompson 2006], and thus there is vast research on code smells and refactoring strategies for specific contexts and languages, such as mobile applications [Habchi et al. 2017], Web development [Ferreira and Valente 2023], video games [Nardone et al. 2023], Python [Zhang et al. 2024], and quantum computing [Chen et al. 2023]. However, the majority of these studies are focused on improving the quality of object-oriented systems [Abid et al. 2020, Sobrinho et al. 2021].

Historically, functional languages have not been as popular in the industry as object-oriented ones. However, there has been a recent increase in interest in functional languages [Bordignon and Silva 2020]. More specifically, Elixir is a modern functional programming language that is gaining traction in the industry, with over 300 companies worldwide using the language, including Adobe, Discord, Heroku, PepsiCo, Pinterest, and some Brazilian companies such as Stone, Rebase, and FinBits.[1] This language is renowned for its performance in parallel and distributed computing environments [Thomas 2018]. Conceived in 2012, Elixir draws inspiration from a blend of programming languages, such as Ruby, Haskell, Erlang, and Clojure [Jurić 2024]. According to approximately 72% of developers who participated in the Elixir Survey 2023,[2] the three main factors that influenced their decisions to adopt Elixir are its functional paradigm, increased productivity, and facilitated support for concurrency.

The recent results of the Stack Overflow Survey[3] also highlight how Elixir is a relevant language in the industry today. According to the last three editions of this survey (*i.e.,* 2022, 2023, and 2024), Elixir is the second most admired programming language among developers, just after Rust. This suggests that a large number of developers who currently use the language intend to continue using it in the coming years. Additionally, according to these surveys, Phoenix—the main framework for web development with Elixir—is currently the most loved web technology. Finally, the Stack Overflow Survey 2024 shows that Elixir developers are the second highest-paid in the industry, only behind Erlang developers—the language that most influenced Elixir's design [Thomas 2018]. We provide an overview of various syntactic and semantic aspects of the Elixir functional language in Chapter 2 of the thesis [Vegi 2024].

[1] `https://elixir-companies.com/`
[2] `https://curiosum.com/surveys/elixir-2023`
[3] `https://insights.stackoverflow.com/survey`

Although Elixir is becoming particularly popular and relevant in the industry, to the best of our knowledge, no study has yet investigated code smells or refactorings specifically tailored for this language. However, just as in any programming language, **it is natural to expect that Elixir developers will make bad design choices and then implement sub-optimal code structures, making their systems challenging to maintain, comprehend, modify, and test. Thus, we also expect these developers to pursue design improvements within their codebase through refactoring strategies**.

Therefore, considering the growing significance of functional languages, particularly Elixir, and the lack of studies regarding the quality of systems developed with this language, this thesis aims to fill this research gap. The main reason for this gap is that existing literature predominantly focuses on the object-oriented paradigm and more traditional programming languages. Therefore, taking inspiration from Fowler's book [Fowler and Beck 1999] but applied in another context, **the general objective of this Ph.D. thesis is to prospect, study, document, evaluate, and correlate code smells and refactoring strategies specifically tailored to the Elixir functional language.**

To achieve this objective, we divided the thesis into three major working units:

1. First, we cataloged 35 Elixir code smells from multiple content sources, such as grey literature, open-source projects, and developer feedback. These code smells were validated by 181 developers who work with Elixir.
2. In the second working unit, we cataloged 82 refactoring strategies specifically aimed at improving the quality of systems developed in Elixir and validated them with 151 developers. For each of these strategies, we provided before-and-after code examples and side conditions to ensure behavior preservation.
3. Finally, in the third working unit, we mapped relationships between Elixir-specific code smells and refactorings, providing developers with practical guidance on using refactorings to systematically eliminate code smells and improve code quality.

Beyond this introduction, the extended abstract has three sections: Section 2 details our research methods, Section 3 presents the main results, and Section 4 summarizes conclusions, contributions, Elixir developers' community impact, and future work.

2. Research Method

In the first working unit of the thesis, aiming to build a catalog of code smell for Elixir, we *first* conducted a qualitative study, extracting and cataloging code smells for Elixir from 17 grey literature documents, 25 documents created by interactions with the Elixir community in GitHub (13 issues and 12 pull requests), and 46 artifacts mined from Elixir repositories on GitHub. *Second*, we validated this catalog of smells by surveying 181 experienced Elixir developers from 37 countries across all continents. Each participant received a list of smells and they were asked to rank each one's relevance and prevalence on a scale of one (*very low*) to five (*very high*). Figure 1 summarizes the steps we followed to propose the catalog of code smells for Elixir.

In the second working unit, with the goal of cataloging refactoring strategies tailored for Elixir, we *first* conducted a systematic literature review where we analyzed 135 research papers to identify and catalog refactoring strategies that have been proposed for other functional languages but that are also compatible with Elixir. *Second*, we cataloged

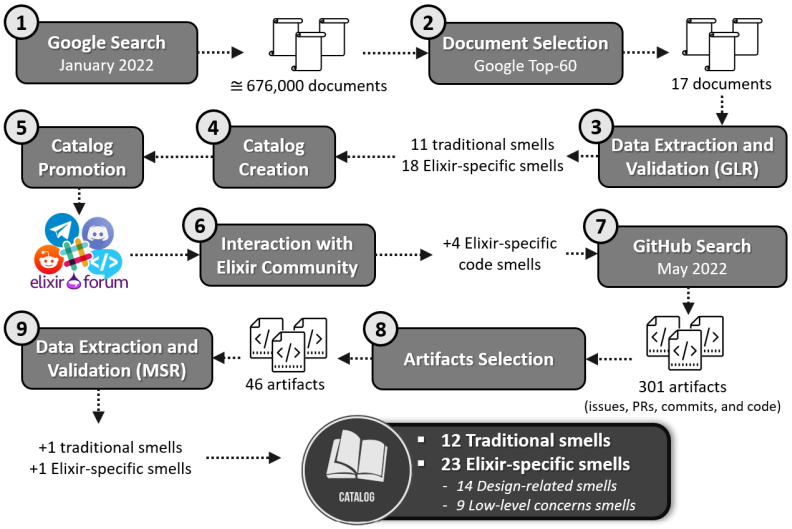

Figure 1. Overview of methods for cataloging code smells in Elixir

refactorings for Elixir from 26 grey literature documents. *Third*, we mined 119 artifacts from the Top-10 Elixir repositories with the most stars on GitHub to expand our catalog of refactorings. *Fourth*, we surveyed 151 experienced Elixir developers from 42 countries to validate this catalog. Similar to what was done for code smells, each participant received a list of refactorings and rated their relevance and prevalence on a scale from one (*very low*) to five (*very high*). Both surveys (*i.e.,* about smells and refactorings) were approved by the Research Ethics Committee at the UFMG prior to their conduction. Figure 2 summarizes our steps to propose the catalog of refactorings for Elixir.

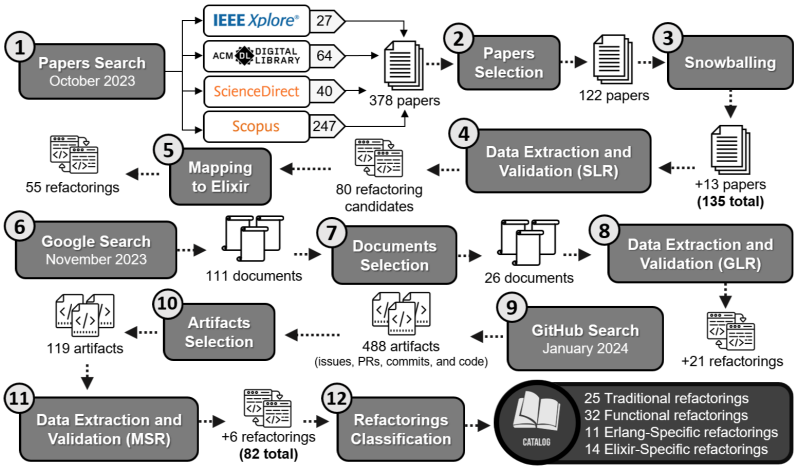

Figure 2. Overview of methods for cataloging refactorings in Elixir

In previous working units, we have cataloged code smells and refactorings specifically tailored for Elixir, but we did not establish direct correlations between them. Therefore, intending to correlate these catalogs in the same way [Fowler and Beck 1999] did for theirs, and thus to create a practical guide on how to remove each code smell in a disciplined way, in the third working unit we *first* conducted an empirical study where each of the 35 code smells previously cataloged by us was manually compared with each of the 82 refactorings. Through these comparisons, we identified the refactorings that could

aid in removing each smell in Elixir and the order in which they should be performed. *Second*, we classified the motivations behind each refactoring not mapped to removing Elixir smells, aiming to understand the reasons for these mapping absences. Figure 3 summarizes the steps we followed to correlate code smells and refactorings for Elixir.

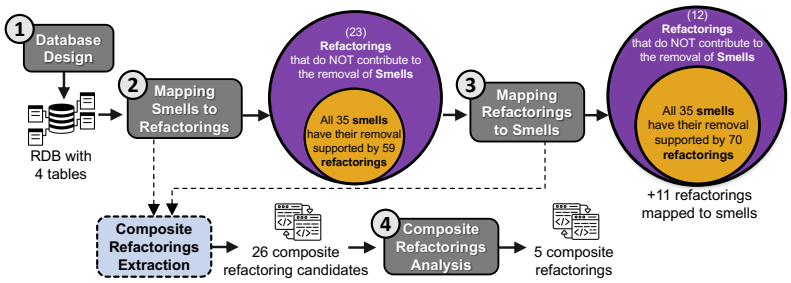

Figure 3. Overview of methods for correlating smells and refactorings in Elixir

All the methods used in this research, including those mentioned above, are detailed in Chapters 3-5 of the thesis [Vegi 2024].

3. Results

In the first working unit of the thesis, we initially proposed a **comprehensive catalog of 35 code smells for Elixir**. This catalog includes 23 novel Elixir-specific code smells and 12 traditional ones (as proposed by [Fowler and Beck 1999]) that also occur in Elixir systems. We categorized the Elixir-specific smells into two groups: nine LOW-LEVEL CONCERNS SMELLS, which have a narrow scope and affect small code structures, and 14 DESIGN-RELATED SMELLS, which are more complex and related to code organization, thus impacting larger portions of code. Each of these smells has been documented with a name, category, problem description, code examples, and suggested transformations to improve smelly code. This information is available in a public GitHub repository (`https://github.com/lucasvegi/Elixir-Code-Smells`).

The `with` statement is an Elixir-specific conditional used for chaining pattern matches. It sequentially compares expressions with patterns, returning a predefined value if all patterns match, or the first non-matching result [Jurić 2024]. COMPLEX ELSE CLAUSES IN WITH is a LOW-LEVEL CONCERNS code smell that refers to `with` statements that group all error clauses into a complex `else` block. Listing 1 presents an example where the function `open_decoded_file/1` reads and decodes a base64 string from a file (lines 2-4). In this function, the `with` statement handles two errors within a single `else` block (lines 5-8). As the function evolves, additional error conditions may be added, which can make the code more difficult to read and maintain.

Listing 1. Example of COMPLEX ELSE CLAUSES IN "WITH"

```
1  def open_decoded_file(path) do
2    with {:ok, encoded} <- File.read(path),
3         {:ok, value} <- Base.decode64(encoded) do
4      value
5    else
6        {:error, _} -> :badfile
7        :error -> :badencoding
8    end
9  end
```

Due to space limitations, only one code smell was presented in this section. However, more details and examples on all smells can be found in Chapter 3 of the thesis [Vegi 2024] and in our catalog on GitHub.

In the first working unit of the thesis, we also showed that 97% of the cataloged Elixir smells have at least mid-relevance, indicating their potential to hinder the readability, maintenance, or evolution of Elixir systems. Additionally, 54% of these smells have at least mid-prevalence, making them common in production code. These findings highlight practical implications for developers and researchers, including prioritizing code smell prevention and removal, advancing tools for automatic smell detection in Elixir, and identifying open research areas related to the catalog.

In the second working unit of this thesis, we proposed a **comprehensive catalog of 82 refactorings for Elixir**. Based on the programming features used by these code transformations, they were categorized into four groups: 14 ELIXIR-SPECIFIC REFACTORINGS, 32 FUNCTIONAL REFACTORINGS, 11 ERLANG-SPECIFIC REFACTORINGS, and 25 TRADITIONAL REFACTORINGS.

The descriptions of all refactorings and more details about each one, including code examples, are available in Chapter 4 of the thesis [Vegi 2024] and at `https://github.com/lucasvegi/Elixir-Refactorings`. The refactoring strategy PIPELINE USING "WITH" is one of the most prevalent and relevant in our catalog, according to the Elixir developers who participated in our survey, making it a representative example to illustrate the main characteristics of the catalog. This is a code transformation that depends on the `with` statement, previously explained. When conditional statements, such as `if..else` and `case`, are nested to control sequences of function calls, the code's readability can become compromised. In these situations, we can replace nested conditionals with a kind of pipeline using a `with` statement, thus performing pattern matching at each function call and interrupting the pipeline if any pattern does not match.

Listing 2 exemplifies an opportunity to apply this refactoring. The function `update_game_state/3` uses nested conditional statements—`if` (line 5) and `case` (line 7)—to ensure the safe invocation of the next function in the sequence: `valid_move/2` (line 4), `players_turn/2` (line 6), and `play_turn/3` (line 8).

Listing 2. Example of a code before PIPELINE USING "WITH"

```
1  # Before refactoring:
2
3  defp update_game_state(%{status: :started} = state, index, user_id) do
4    {move, _} = valid_move(state, index)
5    if move == :ok do
6      players_turn(state, user_id)
7      |> case do
8        {:ok, marker} -> play_turn(state, index, marker)
9        other         -> other
10     end
11   else
12     {:error, :invalid_move}
13   end
14 end
```

Listing 3 shows the result of refactoring `update_game_state/3` using PIPELINE USING "WITH". Using this transformation, the calls to `valid_move/2` (line 4), `players_turn/2` (line 5), and `play_turn/3` (line 6) were chained using pattern matching in each of the three clauses of a `with` statement, thus eliminating the need to

use nested conditional statements. The resulting code from this refactoring also represents an opportunity to perform another refactoring from our catalog—REMOVE REDUNDANT LAST CLAUSE IN "WITH"—as shown in Chapter 4 of the thesis [Vegi 2024].

Listing 3. Example of a code after PIPELINE USING "WITH"

```elixir
# After refactoring:

defp update_game_state(%{status: :started} = state, index, user_id) do
  with {:ok, _}         <- valid_move(state, index),
       {:ok, marker}    <- players_turn(state, user_id),
       {:ok, new_state} <- play_turn(state, index, marker) do
    {:ok, new_state}
  else
    (other -> other)
  end
end
```

Analysis of the survey data from the second working unit of the thesis revealed that 70.6% of the cataloged Elixir refactorings are at least moderately prevalent, indicating their frequent use in production code. Additionally, 92.7% of refactorings are at least moderately relevant, suggesting they can improve Elixir system quality. These findings suggest that developers should first focus on mastering the most prevalent refactorings to streamline the code review process. Conversely, when maintaining Elixir systems, developers should prioritize the most relevant refactorings to maximize code quality improvements.

Finally, in the third working unit of the thesis, we **correlated the cataloged code smells and refactorings for Elixir** and proposed **practical guidelines for removing each code smell in a disciplined manner using refactoring strategies in this language**. In total, we identified 176 relationships between code smells and corresponding refactorings for Elixir, which can be applied to eliminate these smells. Details about each of these relationships between smells and refactorings can be found in Appendix D of the thesis [Vegi 2024] or at `https://doi.org/10.5281/zenodo.14990421`.

More specifically, we found that all 35 Elixir smells have their removal assisted by at least one refactoring also cataloged for this language. Additionally, some refactorings can resolve multiple Elixir smells, while 12 of the 82 cataloged refactorings are not linked to any known Elixir smell. Through these correlations, we identified five new composite refactorings[4] that are effective for eliminating code smells in Elixir.

Furthermore, the results from the third working unit of this thesis showed that traditional refactorings, initially proposed for object-oriented systems [Fowler and Beck 1999], are also effective in eliminating Elixir code smells. Specifically, 51.14% of the 176 mapped relationships involved these refactorings, making them the most influential for code smell removal. These findings provide valuable guidance for developers, especially those new to Elixir, on systematically improving code quality.

All the results found in this research, including those mentioned above, are detailed in Chapters 3-5 of the thesis [Vegi 2024].

[4]Composite refactorings are higher-level transformations, consisting of sequences of atomic refactorings as proposed by [Fowler and Beck 1999].

4. Concluding Remarks

This section discusses the main scientific contributions, academic/professional impact, and future work.

4.1. Scientific Contributions

In summary, the first study of the thesis [Vegi 2024], presented in Chapter 3, contributed by cataloging 23 novel Elixir-specific code smells. We also identified and cataloged 12 traditional code smells, as proposed by [Fowler and Beck 1999], that appear in Elixir systems. Our survey results show that 97% of the cataloged smells have at least mid-relevance, potentially impacting readability, maintainability, or evolution, while 54% exhibit at least mid-prevalence, making them common in production code.

Conversely, the main contributions of the second study of the thesis, presented in Chapter 4, include cataloging 82 Elixir refactorings, 14 of which are novel and specific to this language. We also provided documentation with code examples and side conditions to support the future development of automated refactoring tools for Elixir. Furthermore, our survey results revealed that 70.6% of these refactorings are at least moderately prevalent in production code, and 92.7% are at least moderately relevant, potentially improving Elixir system quality. Additionally, 11% of the refactorings were deemed particularly important due to their high relevance and prevalence. Lastly, we found that the experience levels of developers had minimal impact on their perceptions of refactoring relevance and prevalence, with only 19% of perceptions influenced by these demographic factors.

Finally, the main contribution of the third study of the thesis, presented in Chapter 5, was to provide a detailed explanation of how the removal of each of the 35 Elixir code smells can be assisted by refactoring strategies cataloged for this language. In addition to listing the refactorings useful for removing each smell and explaining their specific applications, we also document the order in which these refactorings should be performed when part of a sequence of operations. This disciplined way to refactor a smell can guide developers to change their code one small step at a time, minimizing the chances of introducing bugs or altering the original behavior of the system.

4.2. Academic and Professional Impact

The thesis [Vegi 2024] resulted in four international publications in highly relevant and impactful venues:

- **ICPC'22** Vegi, L. F. M. and Valente, M. T. (2022). Code smells in Elixir: early results from a grey literature review. In *30th International Conference on Program Comprehension (ICPC) - ERA track*, pages 580–584.
- **EMSE'23** Vegi, L. F. M. and Valente, M. T. (2023b). Understanding code smells in Elixir functional language. *Empirical Software Engineering*, 28(102):1–32.
- **ICSME'23** Vegi, L. F. M. and Valente, M. T. (2023a). Towards a catalog of refactorings for Elixir. In *39th International Conference on Software Maintenance and Evolution (ICSME) - NIER track*, pages 358–362.
- **EMSE'25** Vegi, L. F. M. and Valente, M. T. (2025). Understanding refactorings in Elixir functional language. *Empirical Software Engineering*, 30(108):1–58.

In addition to the scientific contributions from the three working units mentioned earlier, this Ph.D. thesis had some impact on the Elixir Developers' Community:

- Our GitHub repository for cataloging Elixir code smells gained popularity, receiving approximately 1.5k stars and ranking among the 60 most-starred Elixir projects. Due to community interest, **part of this work was later integrated into the official Elixir documentation in collaboration with the core team**.[5]
- Our GitHub repository for cataloging Elixir refactorings has around 170 stars. While not as popular as the code smells repository, it has inspired developers to create tools for automating some cataloged refactorings, such as RefactorEx.[6]
- Since mid-2022, this thesis has been featured in major podcasts for Elixir developers and software engineers, including *Elixir em Foco*,[7] *Thinking Elixir*,[8] *Elixir Mentor*,[9] and *Fronteiras da Engenharia de Software*.[10]
- Since 2023, this thesis has been a recurring topic at Elixir conferences, including ElixirConf 2023[11] and Code BEAM Europe 2023.[12] The author of this thesis also presented a talk on code smells and refactorings at Elixir Fortaleza Conf 2023.[13]
- Finally, José Valim, Elixir's creator, highlighted this thesis in his keynote at ElixirConf EU 2024.[14]

4.3. Future Work

This thesis uncovered several areas for future research: **(i)** evaluating the compatibility of classical metrics for detecting traditional code smells with the detection of Elixir-specific smells and proposing new ones; **(ii)** developing or adapting Elixir-smell detection tools; **(iii)** creating automated refactoring tools for Elixir code; **(iv)** compiling a catalog of composite refactorings for Elixir; **(v)** investigating behavior preservation in refactorings for Elixir; **(vi)** exploring the interrelation between Elixir code smells; **(vii)** measuring refactoring impacts on Elixir software quality; **(viii)** generalizing our smell and refactoring catalogs for the functional programming paradigm.

References

Abid, C., Alizadeh, V., Kessentini, M., Ferreira, T. N., and Dig, D. (2020). 30 years of software refactoring research: a systematic literature review. *ArXiv*, abs/2007.02194:1–23.

Bordignon, M. D. and Silva, R. A. (2020). Mutation operators for concurrent programs in Elixir. In *21st IEEE Latin-American Test Symposium (LATS)*, pages 1–6.

Chen, Q., Câmara, R., Campos, J., Souto, A., and Ahmed, I. (2023). The smelly eight: An empirical study on the prevalence of code smells in Quantum Computing. In *45th IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 1–13.

Ferreira, F. and Valente, M. T. (2023). Detecting code smells in React-based web apps. *Information and Software Technology*, 155:1–16.

[5] https://hexdocs.pm/elixir/what-anti-patterns.html
[6] https://github.com/gp-pereira/refactorex
[7] https://youtu.be/dp8zQUadDgQ
[8] https://podcast.thinkingelixir.com/93
[9] https://youtu.be/BAchf-VSOhY
[10] https://youtu.be/LkBd-x9OLcE
[11] https://youtu.be/aSOY70vydp4
[12] https://youtu.be/6r5b574ttV8
[13] https://youtu.be/klubcNmv4qI
[14] https://youtu.be/agkXUp0hCW8

Fontana, F. A., Ferme, V., Marino, A., Walter, B., and Martenka, P. (2013). Investigating the impact of code smells on system's quality: an empirical study on systems of different application domains. In *29th IEEE International Conference on Software Maintenance (ICSM)*, pages 260–269.

Fowler, M. and Beck, K. (1999). *Refactoring: improving the design of existing code*. Addison-Wesley, 1 edition.

Habchi, S., Hecht, G., Rouvoy, R., and Moha, N. (2017). Code smells in iOS apps: how do they compare to Android? In *4th IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 110–121.

Jurić, S. (2024). *Elixir in action*. Manning, 3 edition.

Lehman, M. (1980). Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076.

Li, H. and Thompson, S. (2006). Comparative study of refactoring Haskell and Erlang programs. In *6th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 197–206.

Meyer, B. (1997). *Object-oriented software construction*. Prentice Hall, 2 edition.

Nardone, V., Muse, B. A., Abidi, M., Khomh, F., and Penta, M. D. (2023). Video game bad smells: What they are and how developers perceive them. *ACM Trans. Softw. Eng. Methodol.*, 32(4):1–35.

Sobrinho, E., De Lucia, A., and Maia, M. (2021). A systematic literature review on bad smells–5 w's: which, when, what, who, where. *IEEE Trans. Softw. Eng.*, 47(1):17–66.

Taibi, D., Janes, A., and Lenarduzzi, V. (2017). How developers perceive smells in source code: a replicated study. *Information and Software Technology*, 92(1):223–235.

Thomas, D. (2018). *Programming Elixir - 1.6: functional - concurrent - pragmatic - fun*. Pragmatic Bookshelf, 1 edition.

Vegi, L. F. M. (2024). *Code Smells and Refactorings for Elixir*. Phd thesis, Universidade Feredal de Minas Gerais, Brazil.

Vegi, L. F. M. and Valente, M. T. (2022). Code smells in Elixir: early results from a grey literature review. In *30th International Conference on Program Comprehension (ICPC) - ERA track*, pages 580–584.

Vegi, L. F. M. and Valente, M. T. (2023a). Towards a catalog of refactorings for Elixir. In *39th International Conference on Software Maintenance and Evolution (ICSME) - NIER track*, pages 358–362.

Vegi, L. F. M. and Valente, M. T. (2023b). Understanding code smells in Elixir functional language. *Empirical Software Engineering*, 28(102):1–32.

Vegi, L. F. M. and Valente, M. T. (2025). Understanding refactorings in Elixir functional language. *Empirical Software Engineering*, 30(108):1–58.

Zhang, Z., Xing, Z., Zhao, D., Xu, X., Zhu, L., and Lu, Q. (2024). Automated refactoring of non-idiomatic Python code with pythonic idioms. *IEEE Trans. on Softw. Eng.*, pages 1–22.