Code Smells and Refactorings for Elixir

A Ph.D. thesis about Software Quality, defended in December 2024 at UFMG

Lucas Francisco da Matta Vegi
Department of Computer Science (DCC), Federal
University of Minas Gerais (UFMG)
Belo Horizonte-MG, Brazil
lucasvegi@dcc.ufmg.br

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

ABSTRACT

Elixir is a modern functional programming language that is steadily gaining popularity in the industry. However, there is still limited research on the internal code quality produced using this language. To address this gap, this Ph.D. thesis explores code smells and refactorings specific to Elixir, taking inspiration from Fowler's classic catalogs. The first two studies employed a mixed-methods approach to identify and catalog 35 code smells (including 23 new ones and 12 commonly known), validated by 181 developers, as well as 82 refactorings (14 of which are novel), validated by 151 developers. The third study mapped relationships between code smells and corresponding refactorings, proposing practical guidelines for their systematic elimination. These findings had a real impact on the community of developers working with the language, contributing both to the prevention of code smells and to supporting the prioritization of refactorings in Elixir development.

KEYWORDS

Code smells, Refactoring, Elixir, Functional programming, Mining software repositories, Grey literature review, Systematic literature review

1 Introduction

Ensuring product quality is a key concern in engineering disciplines, including software engineering. Software quality can be assessed in terms of *external* (e.g., usability, efficiency) and *internal* attributes (e.g., maintainability, readability) [15]. Over time, repeated maintenance degrades internal quality [13], as noted by Fowler [9], who attributes a project failure to excessive code complexity.

Fowler's experience led to a seminal catalog of 72 object-oriented refactorings and 22 code smells [9]. These smells indicate poor design and opportunities for improvement. However, developers' perception of code smells may vary by domain [8, 19], and each language introduces specific challenges [14]. As a result, several studies explore smells and refactorings in diverse contexts, such as mobile apps [10, 11], Web development [7, 17], games [3, 16], and languages like Java [6], Python [26, 27], Ruby [4], and quantum computing [28]—most focused on object-oriented systems [1, 18].

Although traditionally less popular, functional languages are gaining traction in the industry [2]. Elixir, a modern functional language inspired by Ruby, Haskell, Erlang, and Clojure [12], is now adopted by over 300 companies worldwide, including Adobe, Discord, and Brazilian companies like Stone and Rebase.[1] Known for performance in distributed systems [20], in recent Stack Overflow surveys,[2] Elixir ranked as the second most admired language, Phoenix[3]—a well-known framework of the language—became the most loved web framework, and Elixir developers are among the highest paid.

Despite its growing relevance, there is no prior research on code smells or refactorings tailored to Elixir. As with any language, Elixir developers are susceptible to poor design choices and must refactor code to improve maintainability and testability, for example. Thus, this Ph.D. thesis fills this research gap by addressing the absence of studies on internal quality in Elixir systems.

Our goal is to prospect, document, evaluate, and correlate code smells and refactorings specific to Elixir, drawing inspiration from Fowler [9], but adapted to a functional setting. The thesis is organized into three main working units:

(1) We cataloged 35 Elixir code smells from grey literature, open-source code, and direct interactions with developers. These were validated by 181 Elixir developers.
(2) We documented 82 refactorings for Elixir, with before-and-after examples and side conditions, validated by 151 developers.
(3) We mapped the relationships between the smells and refactorings, offering practical guidance to support disciplined code improvements.

Besides this introduction, Section 2 presents the research methods, Section 3 details the main results, Section 4 presents some possibilities for future work, and Section 5 presents the full version of the paper that describes the main aspects of this thesis, as well as references to its key scientific and practical contributions.

2 Research Method

In the first, to build a catalog of Elixir code smells, we *first* performed a qualitative study extracting smells from 17 grey literature documents, 25 GitHub community artifacts (issues and pull requests), and 46 mined code artifacts. *Second*, we validated this catalog via a survey with 181 experienced Elixir developers from 37 countries, who rated each smell's relevance and prevalence on a 1-to-5 scale. Figure 1 summarizes this process.

In the second unit, targeting Elixir refactorings, we *first* conducted a systematic literature review of 135 papers to identify refactorings from functional languages compatible with Elixir. *Second*, we collected refactorings from 26 grey literature sources and *third*, mined 119 artifacts from the top-10 starred Elixir GitHub repositories. *Fourth*, we validated the catalog with 151 Elixir developers

[1]https://elixir-companies.com/
[2]https://insights.stackoverflow.com/survey
[3]https://phoenixframework.org/

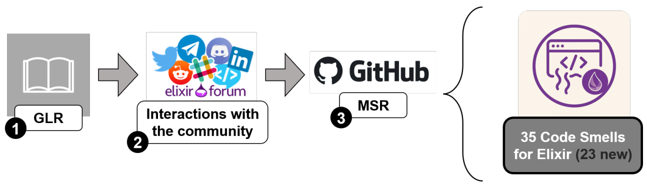
GLR
Interactions with the community
elixir forum
GitHub
MSR
35 Code Smells for Elixir (23 new)

Figure 1: Overview of methods for cataloging code smells

from 42 countries through a similar survey. Both surveys were approved by the Research Ethics Committee at UFMG. Figure 2 illustrates key steps in constructing our refactorings catalog.

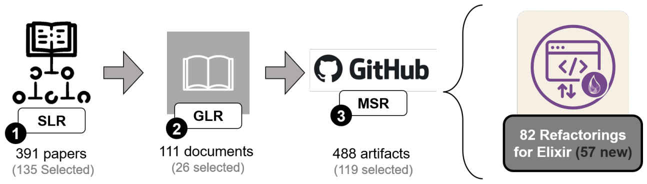

Figure 2: Overview of methods for cataloging refactorings

In the third unit, aiming to correlate smells and refactorings as Fowler [9] did, we *first* manually mapped each of the 35 smells to one or more of the 82 refactorings, defining applicable refactorings to remove each smell for Elixir and their recommended order in a sequence. *Second*, we classified motivations for refactorings not linked to smells to understand these gaps.

Detailed descriptions of these methods are provided in Chapters 3–5 of the thesis [5].

3 Results Overview

In the first working unit, we proposed a **catalog of 35 code smells for Elixir**, including 23 novel Elixir-specific and 12 traditional smells (as proposed by [9]). Elixir-specific smells were grouped into Low-Level Concerns (9) and Design-Related (14). Each smell was documented with name, category, problem description, examples, and suggestions for improvement. The complete catalog is publicly available at https://github.com/lucasvegi/Elixir-Code-Smells.

The `with` construct is a conditional feature unique to Elixir, designed for chaining multiple pattern matches. It evaluates a sequence of expressions against specified patterns, returning a predefined result if all matches succeed, or the value of the first failing match otherwise [12]. Optionally, a `with` expression can include an `else` clause to handle cases where pattern matching fails, which may introduce a code smell. For instance, Listing 1 illustrates the Complex else clauses in "with" smell, where consolidating the handling of multiple error scenarios within a single `else` clause negatively affects both readability and maintainability.

Through the results of the survey conducted in this first unit to validate the smell catalog, we also showed that 97% of the smells are at least moderately relevant, and 54% are at least moderately prevalent in Elixir systems, suggesting both practical and research implications. More details about the survey results and implications, as well as examples of all smells, can be found in Chapter 3 of the thesis [5] and in our catalog on GitHub.

Listing 1: Example of Complex else Clauses in "with"

```
def open_decoded_file(path) do
  with {:ok, encoded} <- File.read(path),
       {:ok, value} <- Base.decode64(encoded) do
    value
  else
      {:error, _} -> :badfile
      :error -> :badencoding
  end
end
```

In the second working unit, we proposed a **catalog of 82 refactorings** for Elixir, classified into 14 Elixir-Specific, 32 Functional, 11 Erlang-Specific, and 25 Traditional refactorings. Full descriptions, examples, and categories are provided in Chapter 4 of the thesis [5] and online at https://github.com/lucasvegi/Elixir-Refactorings.

One representative refactoring is Pipeline using "with", which replaces nested conditionals with a sequence of pattern-matching clauses, maintaining the original behavior of the code while improving code clarity. Listings 2 show a before-and-after example of using this transformation strategy in the `update_game_state/3` function.

Listing 2: Example of using the Pipeline using "with"

```
# Before refactoring:

defp update_game_state(state, index, user_id) do
  {move, _} = valid_move(state, index)
  if move == :ok do
    players_turn(state, user_id)
    |> case do
      {:ok, marker} -> play_turn(state,index,marker)
      other         -> other
    end
  else
    {:error, :invalid_move}
  end
end
```

```
# After refactoring:

defp update_game_state(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
```

Analyzing the results of the survey that validated the catalog of refactorings for Elixir created in this second working unit of the thesis, we found that 70.6% of the refactorings are at least moderately prevalent and 92.7% are at least moderately relevant, respectively, suggesting that developers should prioritize learning and applying them.

Finally, in the third working unit, we **correlated all 35 code smells with 82 refactorings** to support their disciplined removal. We mapped 176 relationships between code smells and corresponding refactorings for Elixir, which can be applied to eliminate these smells. Detailed descriptions of these relationships can be found in Chapter 5 and Appendix D of the thesis [5], as well as at https://doi.org/10.5281/zenodo.14990421. We also identified five composite refactorings (*i.e.,* , sequences of atomic transformations [9]) used in smell removal. All smells were associated with at least one supporting refactoring, while 12 refactorings were not mapped to any smell. Traditional refactorings were involved in 51.14% of the mappings, reinforcing their relevance even in functional programming contexts.

The results of this third working unit can assist developers, especially those new to Elixir, in systematically improving code quality.

All results from this research, including those mentioned above, are detailed in Chapters 3 to 5 of the thesis [5].

4 Future Work

This thesis opens several avenues for future work: **(i)** evaluating the suitability of classical metrics for detecting Elixir-specific smells and proposing new metrics when necessary; **(ii)** creating or adapting smell detection tools for Elixir; **(iii)** building automated refactoring tools; **(iv)** cataloging composite refactorings; **(v)** studying behavior preservation during Elixir refactorings; **(vi)** analyzing interrelations among Elixir smells; **(vii)** evaluating refactoring effects on software quality; **(viii)** generalizing the catalogs to functional programming.

5 Previous Work Reference

The full version of the paper describing the main aspects of this Ph.D. thesis was published in the CBSoft'25 proceedings [21]. The original thesis and the accepted papers resulting from this research are available at the Institutional Repository of UFMG: the thesis itself [5], and papers published at ICPC 2022 [22], ICSME 2023 [23], EMSE 2023 [24], and EMSE 2025 [25], respectively.

ACKNOWLEDGMENTS

We thank the Elixir developers who participated in our surveys and shared their insights on code smells and refactoring. Special thanks to José Valim for facilitating communication with the Elixir community and mediating financial support from FinBits, Dashbit, and Rebase through the Research with Elixir initiative (http://pesquisecomelixir.com.br/).

REFERENCES

[1] Chaima Abid, Vahid Alizadeh, Marouane Kessentini, Thiago N. Ferreira, and Danny Dig. 2020. 30 years of software refactoring research: a systematic literature review. *ArXiv* abs/2007.02194 (2020), 1–23. doi:10.48550/arXiv.2007.02194

[2] Matheus Deon Bordignon and Rodolfo Adamshuk Silva. 2020. Mutation operators for concurrent programs in Elixir. In *21st IEEE Latin-American Test Symposium (LATS)*. 1–6. doi:10.1109/LATS49555.2020.9093675

[3] Matteo Bosco, Pasquale Cavoto, Augusto Ungolo, Biruk Asmare Muse, Foutse Khomh, Vittoria Nardone, and Massimiliano Di Penta. 2023. UnityLint: A Bad Smell Detector for Unity. In *31st IEEE/ACM International Conference on Program Comprehension (ICPC)*. 186–190. doi:10.1109/ICPC58990.2023.00033

[4] Thomas Corbat, Lukas Felber, Mirko Stocker, and Peter Sommerlad. 2007. Ruby Refactoring Plug-in for Eclipse. In *22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion (OOPSLA)*. 779–780. doi:10.1145/1297846.1297884

[5] Lucas Francisco da Matta Vegi. 2024. *Code Smells and Refactorings for Elixir.* PhD thesis. Universidade Federal de Minas Gerais, Brazil. https://repositorio.ufmg.br/handle/1843/80651

[6] Danny Dig. 2011. A Refactoring Approach to Parallelism. *IEEE Software* 28, 1 (2011), 17–22. doi:10.1109/MS.2011.1

[7] Fabio Ferreira and Marco Tulio Valente. 2023. Detecting Code Smells in React-based Web Apps. *Information and Software Technology* 155 (2023), 1–16. doi:10.1016/j.infsof.2022.107111

[8] Francesca Arcelli Fontana, Vincenzo Ferme, Alessandro Marino, Bartosz Walter, and Pawel Martenka. 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)*. 260–269. doi:10.1109/ICSM.2013.37

[9] Martin Fowler and Kent Beck. 1999. *Refactoring: improving the design of existing code* (1 ed.). Addison-Wesley.

[10] Sarra Habchi, Geoffrey Hecht, Romain Rouvoy, and Naouel Moha. 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)*. 110–121. doi:10.1109/MOBILESoft.2017.11

[11] Geoffrey Hecht, Omar Benomar, Romain Rouvoy, Naouel Moha, and Laurence Duchien. 2015. Tracking the software quality of Android applications along their evolution. In *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 236–247. doi:10.1109/ASE.2015.46

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

[13] M.M. Lehman. 1980. Programs, life cycles, and laws of software evolution. *Proc. IEEE* 68, 9 (1980), 1060–1076. doi:10.1109/PROC.1980.11805

[14] Huiqing Li and Simon Thompson. 2006. Comparative Study of Refactoring Haskell and Erlang Programs. In *6th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*. 197–206. doi:10.1109/SCAM.2006.8

[15] Bertrand Meyer. 1997. *Object-oriented software construction* (2 ed.). Prentice Hall Englewood Cliffs.

[16] Vittoria Nardone, Biruk Asmare Muse, Mouna Abidi, Foutse Khomh, and Massimiliano Di Penta. 2023. Video game bad smells: What they are and how developers perceive them. *ACM Trans. Softw. Eng. Methodol.* 32, 4 (2023), 1–35. doi:10.1145/3563214

[17] Amir Saboury, Pooya Musavi, Foutse Khomh, and Giulio Antoniol. 2017. An empirical study of code smells in JavaScript projects. In *24th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 294–305. doi:10.1109/SANER.2017.7884630

[18] 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

[19] Davide Taibi, Andrea Janes, and Valentina Lenarduzzi. 2017. How developers perceive smells in source code: a replicated study. *Information and Software Technology* 92, 1 (2017), 223–235. doi:10.1016/j.infsof.2017.08.008

[20] Dave Thomas. 2018. *Programming Elixir - 1.6: functional - concurrent - pragmatic - fun* (1 ed.). Pragmatic Bookshelf.

[21] Lucas Francisco da Matta Vegi and Marco Tulio Valente. 2025. Code Smells and Refactorings for Elixir. In *Concurso de Teses e Dissertações em Engenharia de Software (CTD-ES) - XVI Congresso Brasileiro de Software: Teoria E Prática (CBSoft)*. 1–15.

[22] Lucas Francisco Matta Vegi and Marco Tulio Valente. 2022. Code smells in Elixir: early results from a grey literature review. In *30th International Conference on Program Comprehension (ICPC) - ERA track*. 580–584. doi:10.1145/3524610.3527881

[23] Lucas Francisco Matta Vegi and Marco Tulio Valente. 2023. Towards a catalog of refactorings for Elixir. In *39th International Conference on Software Maintenance and Evolution (ICSME) - NIER track*. 358–362. doi:10.1109/ICSME58846.2023.00045

[24] 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

[25] Lucas Francisco Matta Vegi and Marco Tulio Valente. 2025. Understanding refactorings in Elixir functional language. *Empirical Software Engineering* 30, 108 (2025), 1–58. doi:10.1007/s10664-025-10652-y

[26] Zejun Zhang, Zhenchang Xing, Xin Xia, Xiwei Xu, and Liming Zhu. 2022. Making Python code idiomatic by automatic refactoring non-idiomatic Python code with pythonic idioms. In *30th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 696–708. doi:10.1145/3540250.3549143

[27] Zejun Zhang, Zhenchang Xing, Dehai Zhao, Xiwei Xu, Liming Zhu, and Qinghua Lu. 2024. Automated Refactoring of Non-Idiomatic Python Code with Pythonic Idioms. *IEEE Transactions on Software Engineering* (2024), 1–22. doi:10.1109/TSE.2024.3420886

[28] Jianjun Zhao. 2023. On Refactoring Quantum Programs in Q#. In *4th IEEE International Conference on Quantum Computing and Engineering (QCE)*. 169–172. doi:10.1109/QCE57702.2023.10203