Code Smells and Refactorings for Elixir

Lucas Francisco da Matta Vegi and Marco Tulio Valente (advisor)

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.*

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.

The success of this project motivated Fowler to write his aforementioned book on refactorings [Fowler and Beck 1999] to communicate to other developers how to improve the quality of their code. In this book, Fowler cataloged 72 refactorings for object-oriented code, helping to popularize these code transformation techniques. Additionally, Fowler and Beck coined the term *"code smell"* to describe sub-optimal code structures that can harm the evolution of software, characterizing them as opportunities for refactoring. In

total, Fowler's book cataloged 22 code smells, also contributing to the popularization of this concept.

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 [Hecht et al. 2015, Habchi et al. 2017], Web development [Saboury et al. 2017, Ferreira and Valente 2023], video games [Nardone et al. 2023], Java [Dig 2011], Python [Zhang et al. 2024, Zhang et al. 2022], Ruby [Corbat et al. 2007], and quantum computing [Zhao 2023, 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[4]—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].

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 main-**

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

tain, 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 code smells for the Elixir language by extracting these sub-optimal structures from multiple content sources, such as grey literature documents, open-source project codebases, and direct interactions with developers of the language. 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 we also validated these strategies with 151 developers who work with the language. For each of these strategies, we provided examples showing the code before and after the transformations, as well as some side conditions to help preserve the behavior of the refactored code.
3. Finally, in the third working unit, we established relationships between the catalogs of code smells and refactorings specific to systems developed in Elixir. This mapping enabled us to provide practical guidance to developers on which refactoring strategies can be used to remove each code smell in a disciplined way, thereby improving the quality of the code.

Besides this introductory section, this extended abstract is composed of three other sections. Section 2 presents the research method used in the Ph.D. thesis. Section 3 shows the main results obtained. Lastly, Section 4 summarizes the conclusions drawn from this thesis. In addition to describing the main scientific contributions and impacts on the Elixir developers' community, this section also outlines ideas we consider interesting for future investigation.

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.

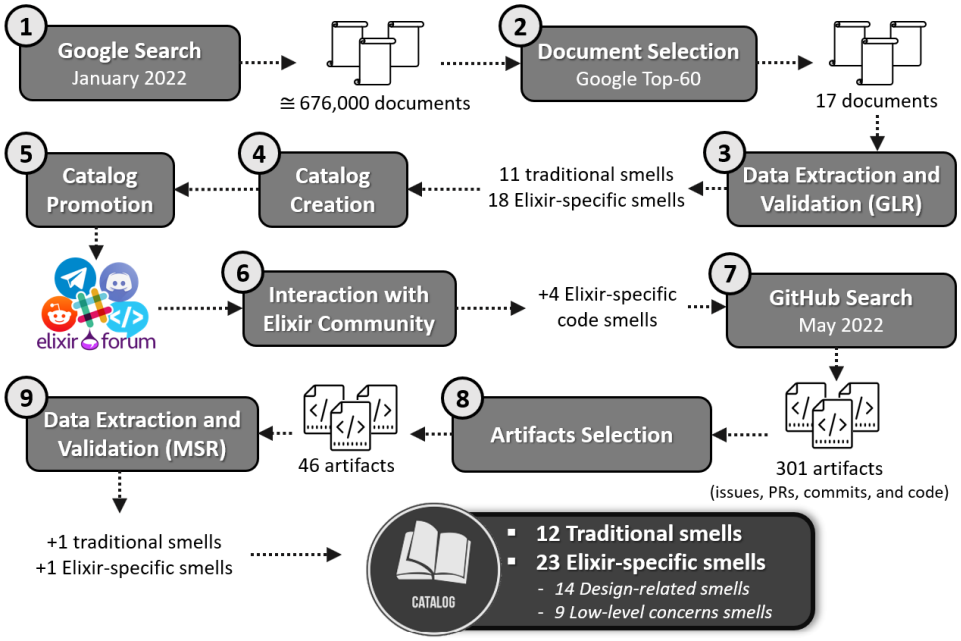

Figure 1. Overview of methods for cataloging code smells in Elixir

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 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 Federal University of Minas Gerais prior to their conduction. Figure 2 summarizes our steps to propose the catalog of refactorings.

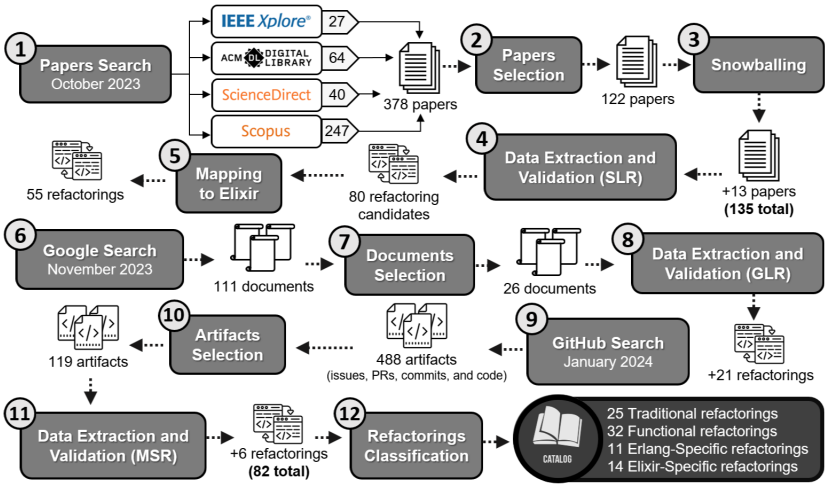

Figure 2. Overview of methods for cataloging refactorings in Elixir

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.

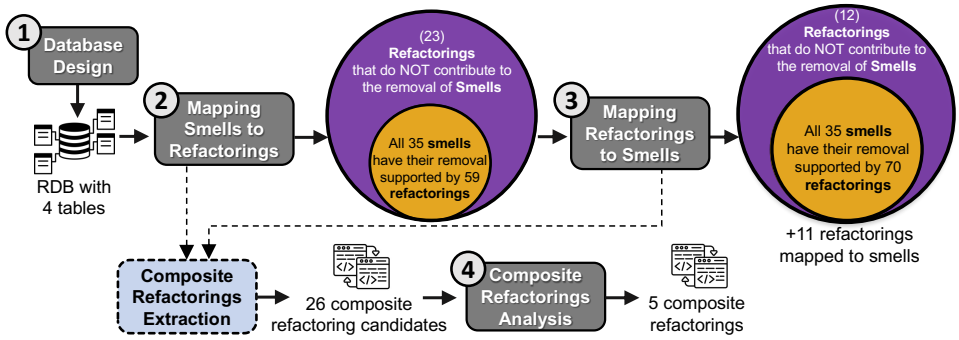

Figure 3. Overview of methods for correlating smells and refactorings in Elixir

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.

Table 1 presents the topics used to document each code smell of our catalog, which is also available in a GitHub public repository (`https://github.com/lucasvegi/Elixir-Code-Smells`).

The `with` statement is an Elixir-specific conditional statement. This conditional is used for pattern matching chaining. It sequentially compares the results of several expressions with patterns, returning a predefined value if all patterns match, or the result of the first expression that does not match a pattern [Jurić 2024]. COMPLEX ELSE CLAUSES IN WITH is a LOW-LEVEL CONCERNS smell that refers to `with` statements that flatten all their error clauses into a single complex `else` block. Listing 1 illustrates an instance of this smell, where the function `open_decoded_file/1` reads a base 64 encoded string content from a file (lines 2-3) and returns a decoded binary string (line 4). This

Table 1. Sections of the catalog of Elixir-specific code smells

Topic	Description
NAME	Unique name of the code smell. This name is important to facilitate communication between developers.
CATEGORY	The portion of code affected by the smell and its granularity.
PROBLEM	How the code smell can harm code quality and the impacts it can have on software maintenance, comprehension, and evolution.
EXAMPLE	Code example and description to illustrate the occurrence of the smell
REFACTORING	Code transformations to change smelly code in order to improve its quality. Examples are also presented to illustrate these changes.

function has an `with` statement that handles two possible errors in a single `else` block (lines 5-8). Although this example has an `else` block that handles only two errors, as this function evolves it can gain new types of errors, which will harm readability and maintainability.

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 [Vegi and Valente 2022a].

Additionally, in the first working unit of the thesis we demonstrated that the majority of cataloged smells (97%) have at least mid-relevance levels, indicating their potential to hinder the readability, maintenance, or evolution of Elixir systems. Furthermore, we showed that most of these smells (54%) exhibit at least mid-prevalence levels, making them common in production code. These findings have practical implications for developers and researchers, such as establishing priorities for preventing and removing code smells, directing efforts toward the evolution of tools for the automatic detection of code smells in Elixir, and identifying open research fields related to the catalog.

In the second working unit of this thesis, we also proposed a **comprehensive catalog of 82 refactorings for Elixir**. To better organize this catalog, we categorized the refactorings into four distinct groups: 14 ELIXIR-SPECIFIC REFACTORINGS, 32 FUNCTIONAL REFACTORINGS, 11 ERLANG-SPECIFIC REFACTORINGS, and 25 TRADITIONAL REFACTORINGS. This classification was done based on the programming features used by the code transformations, as described in Table 2.

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:`

Table 2. Categories used to organize the catalog of refactorings for Elixir

Category	Description	#	%
TRADITIONAL	Refactoring strategies described on Fowler's catalog [Fowler and Beck 1999]	25	30.5
FUNCTIONAL	Refactorings using characteristic features of functional languages (*e.g.,* pattern matching, list comprehension, pipelines, and higher-order functions)	32	39.0
ERLANG-SPECIFIC	Refactorings using features unique of the Erlang ecosystem (*e.g.,* OTP, `typespecs`, and `behaviours`)	11	13.4
ELIXIR-SPECIFIC	Refactorings using specific features of this language (*e.g.,* `with` statements, `Agents`, `Tasks`, and `Streams`)	14	17.1

`//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
```

After analyzing the data collected from the survey conducted in the second working unit of the thesis, we revealed that most of the cataloged refactorings for Elixir (70.6%) are at least moderately prevalent, indicating their recurring use in production code. Furthermore, we showed that the vast majority of refactorings (92.7%) are at least moderately relevant, suggesting they have the potential to enhance the quality of Elixir systems. These findings suggest that developers should first focus on mastering the most prevalent refactorings, as understanding these transformations can streamline the code review process. Conversely, when maintaining their systems and encountering multiple refactoring opportunities, developers should apply the most relevant refactorings first 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 code smells for Elixir have their removal assisted by at least one refactoring also cataloged for this language. Additionally, we showed that some refactorings can resolve multiple code smells, while 12 of the 82 cataloged refactorings are not associated with the removal of any known Elixir smell. Through these correlations, we also identified five new composite refactorings,[5] that are useful for removing code smells in Elixir.

To illustrate the step-by-step removal of a smell through a composite refactoring, we present the case of the smell LARGE CODE GENERATION BY MACROS. Macros are meta-programming mechanisms in Elixir that can extend the language.[6] They enable robust code generation at compile time, which helps reduce boilerplate and allows the creation of DSL constructs for Elixir [Jurić 2024]. The code smell used as an example in this section is related to macros that generate too much code. When a macro generates a large amount of code, it impacts how the compiler or the runtime work. The reason

[5]Composite refactorings are higher-granularity transformations, characterized as sequences of atomic refactorings as those proposed by [Fowler and Beck 1999].

[6]`https://hexdocs.pm/elixir/macros.html`

for this is that Elixir may have to expand, compile, and execute the code multiple times, which makes compilation slower and the compiled artifacts larger.

Listing 4 shows a module used to access a router for a web application. The function `show/0` (line 7) lists all the routes defined for this application. They are stored in the Elixir's module attribute `@store_routes`, which is created and modified by the calls to the macro `Routes.get/2` (lines 4 and 5).

Listing 4. Router for a web application

```
1  defmodule MyApp.Routes do
2    require Routes
3
4    Routes.get("/home", MyApp.HomeController)
5    Routes.get("/about", MyApp.AboutController)
6
7    def show() do
8      @store_routes
9    end
10 end
11 ...
12 iex> MyApp.Routes.show
13 [{"/about", MyApp.AboutController}, {"/home", MyApp.HomeController}]
```

As shown in Listing 5, the macro `get/2` (lines 3 to 19) is an instance of the LARGE CODE GENERATION BY MACROS smell. On every invocation of this macro, which could be hundreds, the code inside `get/2` is expanded and compiled, which can generate a large volume of code overall. This occurs because most of the code defined in `get/2` could be implemented in a conventional function, thus avoiding excessive expansions and compilations.

Listing 5. Instance of the smell LARGE CODE GENERATION BY MACROS

```
1  defmodule Routes do
2    ...
3    defmacro get(route, handler) do
4      quote do
5        route = unquote(route)
6        handler = unquote(handler)
7
8        if not is_binary(route) do
9          raise ArgumentError, "route must be a binary"
10       end
11
12       if not is_atom(handler) do
13         raise ArgumentError, "handler must be a module"
14       end
15
16       Module.register_attribute(__MODULE__, :store_routes, accumulate: true)
17       Module.put_attribute(__MODULE__, :store_routes, {route, handler})
18     end
19   end
20 end
```

This smell can be removed using the composite refactoring EXTRACT TO OUTSIDE. This refactoring consists of the following sequence of atomic refactorings: EXTRACT FUNCTION → MOVE A DEFINITION. Thus, the first step in removing this smell

is to apply EXTRACT FUNCTION. With this refactoring, we can extract part of the macro's code and encapsulate it into a conventional function, which the macro will then call. As shown in Listing 6, by extracting to the function `__define__/3` (line 9) the code originally defined inside the `quote/1` (Listing 5 - lines 4 to 18), we reduce the amount of code that is expanded and compiled on every invocation of `get/2`, and instead we dispatch to `__define__/3` to do the bulk of the work.

Listing 6. Intermediate code version after EXTRACT FUNCTION

```
1  defmodule Routes do
2    ...
3    defmacro get(route, handler) do
4      quote do
5        Routes.__define__(__MODULE__, unquote(route), unquote(handler))
6      end
7    end
8
9    def __define__(module, route, handler) do
10     if not is_binary(route) do
11       raise ArgumentError, "route must be a binary"
12     end
13
14     if not is_atom(handler) do
15       raise ArgumentError, "handler must be a module"
16     end
17
18     Module.register_attribute(module, :store_routes, accumulate: true)
19     Module.put_attribute(module, :store_routes, {route, handler})
20   end
21 end
```

After extracting the function `__define__/3` into the `Routes` module (Listing 6), we can also move it to the `Routes.Utils` module using the MOVING A DEFINITION refactoring, thus making the code more cohesive (Listing 7). This occurs because, although omitted in Listing 7, `Routes.Utils` groups other functions with the same objective as `__define__/3`. As a result of this second step, the code smell LARGE CODE GENERATION BY MACROS is completely removed, since the macro now generates only the minimum necessary amount of code to maintain the original system behavior, and we also achieve cleaner and more organized code.

Listing 7. Code smell is completely removed after using a composite refactoring

```
1  defmodule Routes do
2    ...
3    defmacro get(route, handler) do
4      quote do
5        Routes.Utils.__define__(__MODULE__, unquote(route), unquote(handler))
6      end
7    end
8  end
```

```
1  defmodule Routes.Utils do
2    ...
3    def __define__(module, route, handler) do
4      if not is_binary(route) do
5        raise ArgumentError, "route must be a binary"
6      end
```

```
 7
 8      if not is_atom(handler) do
 9        raise ArgumentError, "handler must be a module"
10      end
11
12      Module.register_attribute(module, :store_routes, accumulate: true)
13      Module.put_attribute(module, :store_routes, {route, handler})
14    end
15  end
```

Furthermore, our results from the third working unit of this thesis demonstrated that traditional refactorings, originally proposed to enhance the quality of object-oriented systems [Fowler and Beck 1999], also play a significant role in eliminating code smells in Elixir. Specifically, 51.14% of the 176 relationships mapped in this study involved refactorings from this category, making it the most influential in code smell removal. The findings of this final working unit can help guide developers—especially those new to Elixir—on how to systematically remove code smells and enhance the internal quality of their systems.

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

4. Concluding Remarks

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

4.1. Scientific Contributions

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

On the other hand, the main contributions of the second study of the thesis, presented in Chapter 4, include cataloging 82 refactorings for Elixir, 14 of which are novel in the scientific literature and specific to this language. We also provided documentation with code examples and tailored side conditions to support the future development of automated refactoring tools for Elixir. Furthermore, our survey results showed that 70.6% of the cataloged refactorings are at least moderately prevalent in production code, and 92.7% are at least moderately relevant, potentially able to enhance 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 the developers in our survey had minimal impact on their perceptions of refactoring relevance and prevalence, as only 19% of the perceptions were 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. (2022b). 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 created to catalog code smells for Elixir became popular among Elixir developers, receiving approximately 1.5k stars, thus ranking among the 60 most-starred Elixir GitHub-based projects.[7] Due to the interest sparked in the developer community for this content, **part of this work was later incorporated into the official Elixir documentation through collaboration with the core team that maintains the language**.[8]
- The GitHub repository we used to catalog refactorings for Elixir has around 171 stars.[9] Although it has not become as popular as our code smells repository, some Elixir developers are already drawing inspiration from it to create tools capable of automatically applying some of the refactoring strategies cataloged in this thesis (*e.g.,* RefactorEx[10]).
- Since mid-2022, the studies conducted in this thesis have been repeatedly discussed in major podcasts aimed at Elixir developers and, more broadly, at software engineers in general. For example, *Elixir em Foco*,[11] *Thinking Elixir*,[12] *Elixir Mentor*,[13] and *Fronteiras da Engenharia de Software*.[14]

[7] https://github.com/lucasvegi/Elixir-Code-Smells
[8] Official documentation: https://hexdocs.pm/elixir/what-anti-patterns.html
[9] https://github.com/lucasvegi/Elixir-Refactorings
[10] https://github.com/gp-pereira/refactorex
[11] Elixir em Foco: https://youtu.be/dp8zQUadDgQ
[12] https://podcast.thinkingelixir.com/93
[13] Elixir Mentor: https://youtu.be/BAchf-VSOhY
[14] Fronteiras da Engenharia de Software: https://youtu.be/LkBd-x9OLcE

- Furthermore, since 2023, this thesis has also been the recurring theme of talks at conferences aimed at Elixir developers, such as ElixirConf 2023[15] and Code BEAM Europe 2023.[16] Additionally, the author of this Ph.D. thesis gave a talk on Elixir-specific code smells and refactorings at Elixir Fortaleza Conf 2023.[17]
- Finally, the importance of the work conducted in this thesis was also recognized by José Valim, the creator of Elixir, who extensively mentioned it in his keynote at ElixirConf EU 2024.[18]

4.3. Future Work

Throughout the research conducted in the thesis, we identified several unexplored topics with the potential for future studies: **(i)** conducting studies aimed at verifying whether classical metrics can be used to detect code smells specific to a functional language like Elixir, and eventually proposing new metrics tailored to this context; **(ii)** creating or adapting tools for detecting code smells in Elixir; **(iii)** creating and adapting tools for automated refactoring in Elixir; **(iv)** proposing a comprehensive catalog of composite refactorings for Elixir; **(v)** conducting studies to ensure the behavior preservation of all Elixir refactorings; **(vi)** investigating the interrelation between code smells for Elixir; **(vii)** quantifying the impacts of refactorings on Elixir software quality attributes; **(viii)** generalizing the catalogs of smells and refactorings for the functional programming paradigm.

Acknowledgments

We thank the Elixir developers who participated in our surveys and shared their perspectives about code smells and refactoring in this language. Finally, José Valim deserves particular thanks for facilitating our communication with the Elixir community and mediating the financial support provided to this research by the companies FinBits, Dashbit, and Rebase (via the Research with Elixir initiative - `http://pesquisecomelixir.com.br/`).

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.

Corbat, T., Felber, L., Stocker, M., and Sommerlad, P. (2007). Ruby refactoring plug-in for Eclipse. In *22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion (OOPSLA)*, pages 779–780.

[15]ElixirConf 2023 talk: `https://youtu.be/aSOY70vydp4`
[16]Code BEAM Europe 2023 talk: `https://youtu.be/6r5b574ttV8`
[17]Elixir Fortaleza Conf 2023 talk: `https://youtu.be/klubcNmv4qI`
[18]ElixirConf EU 2024 keynote talk: `https://youtu.be/agkXUp0hCW8`

Dig, D. (2011). A refactoring approach to parallelism. *IEEE Software*, 28(1):17–22.

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

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.

Hecht, G., Benomar, O., Rouvoy, R., Moha, N., and Duchien, L. (2015). Tracking the software quality of Android applications along their evolution. In *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 236–247.

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 Englewood Cliffs, 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.

Saboury, A., Musavi, P., Khomh, F., and Antoniol, G. (2017). An empirical study of code smells in JavaScript projects. In *24th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 294–305.

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 Transactions on Software Engineering*, 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 Federal de Minas Gerais, Brazil.

Vegi, L. F. M. and Valente, M. T. (2022a). Catalog of Elixir-specific code smells. Available at: `https://github.com/lucasvegi/Elixir-Code-Smells`.

Vegi, L. F. M. and Valente, M. T. (2022b). 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., Xia, X., Xu, X., and Zhu, L. (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)*, page 696–708.

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 Transactions on Software Engineering*, pages 1–22.

Zhao, J. (2023). On refactoring quantum programs in Q#. In *4th IEEE International Conference on Quantum Computing and Engineering (QCE)*, pages 169–172.