RESEARCH ARTICLE - EMPIRICAL

Mind the gap: Linking refactorings and code smells in Elixir

Lucas Vegi[1] | **Marco Túlio Valente**[2]

[1]Department of Computer Science, Federal University of Viçosa (UFV), Viçosa, Brazil

[2]Department of Computer Science, Federal University of Minas Gerais (UFMG), Belo Horizonte, Brazil

Correspondence
Lucas Vegi, Department of Computer Science, Federal University of Viçosa (UFV), Viçosa, Brazil
Email: lucas.vegi@ufv.br

Funding information
Dashbit; Rebase Serviços e Consultoria de Informática Ltda; FAPEMIG.

Elixir is a functional programming language increasingly used in the industry to develop scalable and fault-tolerant concurrent systems more easily and with fewer computational resources. In previous studies, we cataloged 35 code smells and 82 refactorings tailored for this language, validating them with over 300 experienced developers worldwide. Aiming to complement the results of these previous studies, this work manually compares each code smell for Elixir with all the cataloged refactorings for the language, proposing practical guidelines for removing code smells in a disciplined manner within this specific context. In total, we mapped 176 relationships between all 35 code smells and 70 corresponding refactorings that can aid in their elimination. Additionally, we identified five new composite refactorings and found evidence suggesting the existence of an uncataloged code smell for Elixir. Our results were validated through a survey conducted with ten experienced Elixir developers and have practical implications for how code smells can be removed in this language.

KEYWORDS
refactorings, composite refactorings, code smells, Elixir, functional programming, software evolution

Note: For detailed descriptions of all relationships between refactorings and smells referred to in this paper, please consult the Appendix A.

1 | INTRODUCTION

Elixir[1] is a modern functional programming language that has been gaining popularity in the industry in recent years. Currently, more than 300 companies around the world use Elixir in production code, including well-known names such as Pinterest, Adobe, Heroku, Discord, and Motorola[2]. The results of the Stack Overflow Survey 2024[3], which was answered by approximately 65k developers, highlight how Elixir is a relevant language in the industry today. According to this survey, Elixir is the second most admired programming language among developers, just after Rust. Additionally, Phoenix[4]—the main framework for web development with Elixir—is currently the most loved web technology. Furthermore, the latest edition of the Stack Overflow Survey shows that Elixir developers are the second highest-paid in the industry, only behind Erlang developers—the language that most influenced Elixir's design [1]. Haskell, Ruby, and Clojure also served as inspirations for the creation of Elixir [2], leading to the development of an extensible language with a friendly syntax, ideal for building fault-tolerant systems designed for concurrent and distributed environments [3].

Code smells are sub-optimal code structures that can harm software maintenance and evolution [4, 5]. In addition to cataloging 22 of these structures, Fowler and Beck [6] also cataloged 72 disciplined code modification strategies, known as refactorings, and correlated them with the removal of each of these smelly code. The impacts on software quality caused by code smells are not homogeneous and can vary across different domains [7]. Additionally, each programming language has its own constraints and challenges when performing refactorings [8]. For these reasons, there is extensive research on code smells [9, 10, 11, 12, 13] and refactorings [14, 15, 16, 17] for specific contexts and languages. However, the majority of these studies focus on improving the quality of object-oriented systems [18, 19]. Considering the growing interest in Elixir in the industry and the lack of investigations into the quality of code implemented in this language, we have cataloged, in previous studies, 35 code smells [20] and 82 refactorings [21] specifically tailored for Elixir. Although we have proposed and validated these catalogs, to the best of our knowledge, there is still no direct correlation established between these two catalogs in the same way Fowler and Beck [6] did for theirs. To fill this gap, this paper aims to create a practical guide on how to remove a code smell in a disciplined way with the assistance of refactoring strategies in Elixir.

To accomplish this, we conducted an empirical study where each of the 35 code smells previously cataloged by us for Elixir [20] was manually compared with each of the 82 refactorings that we also cataloged for this language [21]. Through these comparisons, we identified the refactorings that could aid in removing each smell and in what order they should be performed. The methodology used in this paper to establish and describe relationships between code smells and refactorings is similar to the one used by Fowler and Beck [6] to correlate their well-known catalogs. Many other authors have also conducted studies that empirically mapped refactoring strategies to eliminate code smells [14, 22, 23, 24, 25, 26, 27, 28]. Our results were validated through a survey answered by ten developers with extensive experience in the Elixir language, drawn from diverse cultural backgrounds, genders, and companies.

The contributions of this paper are fivefold: (1) we found that all 35 code smells for Elixir have their removal assisted by at least one refactoring also cataloged for this language; (2) we identified five new composite refactorings (*i.e.*, sequences of interrelated atomic refactorings [29, 30]) that are useful for removing code smells in Elixir, with three of them working in conjunction with other refactorings to assist in smell removal; (3) we have found evidence suggesting the existence of an uncatalogued smell for Elixir; (4) we showed, through the results of our survey, that

[1] `https://elixir-lang.org/`

[2] Companies using Elixir in production: `https://elixir-companies.com/`

[3] `https://survey.stackoverflow.co/2024/`

[4] `https://phoenixframework.org/`

in the perception of experienced developers, the vast majority of code smells in Elixir (82.9%) can be largely or fully removed through sequences of refactorings tailored to the language. The evaluations demonstrate high inter-rater consistency, providing strong empirical support for the reliability of the reported findings; (5) we showed that traditional refactorings, originally proposed to improve the quality of object-oriented systems, are also highly important for removing code smells in Elixir. The results of this study can guide developers—especially those new to Elixir—on how to systematically remove code smells and enhance the internal quality of their systems built with this language.

The remainder of this paper is structured as follows. Section 2 provides an overview of some important features of the Elixir language. We present relevant related work in Section 3, including our previous studies on code smells and refactorings for Elixir. In Section 4, we detail the methods used to correlate code smells with refactorings that can assist in their removal. Next, in Section 5, we present the results of this mapping, detailing step-by-step how certain code transformation strategies can be used to remove sub-optimal structures. Additionally, in this section, we introduce five new composite refactorings that are useful for removing code smells in Elixir. In Section 6, we delve into the details of the survey conducted with Elixir developers to validate the practical guidelines proposed in this study for removing code smells through sequences of refactorings in the language. In addition to presenting the results of this validation instrument, this section also describes the questionnaire design and the methods used to recruit respondents. In Section 7, we discuss the practical implications of our findings and explore potential reasons for the absence of relationships for some refactorings in our catalog. Potential threats to validity and mitigation strategies are presented in Section 8. Finally, we conclude the paper in Section 9.

2 | ELIXIR FUNCTIONAL LANGUAGE

Elixir is a modern functional programming language created in 2012, inspired by languages such as Haskell, Clojure, Ruby, and Erlang [2]. Although Elixir's syntax is similar to Ruby, it is fully interoperable with Erlang, as both languages run on BEAM, Erlang's virtual machine. More specifically, Elixir programmers can invoke any Erlang function without incurring any runtime cost [1]. BEAM is renowned for its robustness, fault tolerance, low latency, and ability to efficiently handle concurrent and distributed systems [3]. Due to these strengths, Elixir is primarily used for developing high-demand applications, although it can also be used to develop systems for many other purposes, such as machine learning, data pipelines, and multimedia processing.

Elixir programs are organized into `modules`, which are collections of functions. Listing 1 presents an Elixir module (`Foo`) that includes different versions of the recursive function `fibonacci/1` (lines 2 to 6). This grouping of various versions of the same function, known as a multi-clause function in Elixir, is made possible by the pattern matching mechanism [31]. This mechanism is common in functional languages and often serves as a control-flow mechanism in Elixir [2].

As shown in Listing 1, `fibonacci/1` has three clauses with the same name, all accepting an `integer` as a parameter. When `fibonacci/1` is called, these three clauses are matched against the parameter to determine which one will be executed. The Elixir interactive shell (IEx)[5], which is an intelligent terminal that allows developers to run their code and perform various other tasks, was used in this example to call the clause that handles the recursive case (line 9) and the two clauses that handles the base cases (lines 10 and 11) of the function, respectively.

In Elixir, the `when` statement can also be used to define guard clauses, as demonstrated in the recursive case of `fibonacci/1` (line 4). A guard clause enables a function to perform more complex pattern matching checks, determining whether the code within the body of one of its clauses will be executed when the function is called. For instance,

[5] `https://hexdocs.pm/iex/IEx.html`

the use of the `when` statement in the `fibonacci/1` function prevents the recursive case (line 4) from running if a value of a type other than `integer` is provided in the function call, which would otherwise result in an error. This feature is particularly useful for type validation in a dynamically-typed language like Elixir [2].

LISTING 1 Example of code organization and some of Elixir's features

```
1  defmodule Foo do
2    def fibonacci(0), do: 0
3    def fibonacci(1), do: 1
4    def fibonacci(n) when is_integer(n) do
5      fibonacci(n-1) + fibonacci(n-2)
6    end
7  end
8  ...
9  iex(1)> Foo.fibonacci(8) #21
10 iex(2)> Foo.fibonacci(0) #0
11 iex(3)> Foo.fibonacci(1) #1
```

3 | RELATED WORK

In addition to documenting their well-known catalogs of code smells and refactorings, Fowler and Beck [6] establish a relationship between these catalogs, characterizing code smells as opportunities for refactoring. For each of the 22 traditional code smells, the authors suggest refactoring techniques to remove them, which can be either atomic (*i.e.*, not decomposable into simpler refactorings [32]) or composite (*i.e.*, set of these atomic ones [29, 30]).

According to Brito *et al.* [29], composite refactorings are coarse-grained and complex source code transformations consisting of sequences of atomic refactorings. In their work, the authors proposed a catalog of eight strategies, where some composite refactorings comprise sequences of atomic refactorings of the same type (*e.g.*, multiple *Pull Up Method*), while others consist of sequences of atomic refactorings of different types (*e.g.*, multiple *Move* operations eventually followed by a *Rename*, or a sequence of *Extract Method* operations followed by *Move Method* operations). This characteristic regarding the structure of composite refactorings is formalized by Souza *et al.* [30], who define that all refactorings in a composite can either be of the same type or not. Souza *et al.* [30] refer to this as *composite uniformity*, a concept also used in our definition of the five composite refactorings we identified for Elixir.

Similar to the present paper, some studies have also established relationships between code smells and refactorings for specific contexts. Ferreira *et al.* [14] cataloged 69 distinct refactorings employed by developers when maintaining and evolving *React-based* web applications. Additionally, the authors linked these refactorings to 12 *React-related* code smells [12], thereby providing practical guidance for developers on how to remove each smell through refactorings in this specific context. To establish the relationships between code smells and refactorings in the context of React, the first author of that study initially conducted a manual analysis, examining for each *React-related* code smell whether there existed refactoring strategies specific to that context capable of removing it. After the relationships were defined, the results were discussed with the other authors with the aim of refining the mappings and reaching consensus. This methodological procedure is similar to the one adopted in the present study. However, in contrast to the React study, we further complemented the mapping process with an external validation of the results by conducting a survey with ten developers experienced in Elixir.

Regarding the mapping of code smells and refactorings in object-oriented languages, Shetty and Sharma [33] analyzed 212,664 commits from 87 open-source Java projects to investigate how refactoring activities relate to code smell removal. Their study combined automated analysis and manual inspection, using *RefactoringMiner* [34] to iden-

tify refactorings and *DesigniteJava* [35] to detect code smells, two tools widely adopted by the software engineering research community. Although their methodology differs from ours, largely due to the availability of mature automation support for Java that has no direct counterpart for Elixir, some findings converge across both investigations. In particular, both studies identify the *Extract Method/Function* refactoring as one of the most effective strategies for addressing multiple types of code smells. Additionally, they show that not all refactoring activities are necessarily related to code smell removal.

The use of refactorings to address code smells in functional languages has also been explored in the literature. Li *et al.* [22] present a list of refactoring strategies capable of removing seven process-related smells specific to Erlang. The authors also discuss the challenges of automatically performing these refactorings due to Erlang's dynamic nature, the implicit nature of processes, and the communication structure between them in this language. Additionally, Li and Thompson [23] present other refactoring strategies for Erlang that can be semi-automatically performed by the *Wrangler* tool to remove four *Erlang-specific* smells related to the modularity of systems developed in this language. Other studies focus exclusively on refactoring the traditional code smell *Duplicated Code* in Erlang [24, 25, 26, 36] and Haskell [27].

In a previous study [20], we employed a mixed methodology that included a grey literature review, interactions with the Elixir community, and mining GitHub code repositories to catalog 35 code smells that occur in Elixir systems. Of these, 23 are novel and specific to the language. These *Elixir-specific* smells were categorized into *Design-Related smells* (14) and *Low-Level Concerns smells* (9). Additionally, our catalog includes 12 traditional smells, as described by Fowler and Beck [6], which are also present in Elixir systems. Part of this work was later incorporated into the official Elixir documentation through collaboration with the core team that maintains the language[6]. In another previous study [21], we proposed and validated a comprehensive catalog comprising 82 refactorings, including 14 new ones specific to Elixir, 32 aimed at functional languages, 11 Erlang-specific transformations compatible with Elixir, and 25 traditional refactorings cataloged by Fowler [6], which are also applicable to Elixir code. Similar to what we did in our aforementioned study on code smells for Elixir [20], this catalog of refactorings was built based on a systematic literature review, a grey literature review, and mining GitHub code repositories. It was subsequently validated by surveying 144 experienced Elixir developers from 42 countries across all continents.

While our previous studies have independently proposed catalogs of code smells and refactorings for Elixir, no prior work has systematically investigated or validated the relationships between these two artifacts. This paper addresses this gap by establishing an explicit, practitioner-validated mapping between Elixir code smells and refactoring strategies. Our contribution goes beyond cataloging by integrating previously disconnected bodies of knowledge and providing empirically supported refactoring guidance that enables incremental, behavior-preserving code improvements. As a result, developers are better supported in reasoning about quality enhancements while reducing the risk of introducing defects. To the best of our knowledge, this is the first study to deliver a validated smell–refactoring mapping in the context of a functional programming language.

4 | STUDY DESIGN

According to Fowler and Beck [6], while it is important to catalog the mechanics of a refactoring strategy, it is also essential to document when each of these code transformations should be applied. Complementarily, according to Sharma *et al.* [37], merely cataloging the quality issues caused by each code smell is not sufficient for their disciplined removal. Thus, identifying refactorings that effectively remove these smells is equally important.

[6] https://hexdocs.pm/elixir/main/what-anti-patterns.html

Considering the importance of establishing this type of relationship between smells and refactorings, and that this paper is part of a wider research project inspired by Fowler's book [6] to promote quality improvements in systems developed in Elixir, we decided to map the relationships between code smells and refactorings in this language in a manner similar to that carried out by Fowler and Beck [6], thus maintaining coherence between our work and one of its main inspirations. Figure 1 summarizes the steps we followed to correlate code smells and refactorings for Elixir. We also detail all these steps in the following paragraphs.

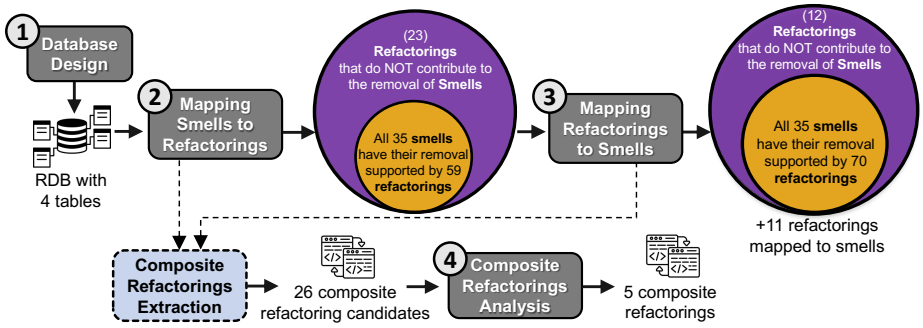

FIGURE 1 Overview of methods for correlating code smells and refactorings in Elixir

1) Database Design: As observed in the mappings between code smells and refactorings conducted by Fowler [6] and also by Ferreira *et al.* [14], the same refactoring strategy can be used in the removal of different types of code smells. Additionally, a code smell can also be removed with the contribution of different refactoring strategies, which may vary according to the problem faced by the developer. Considering this recurring behavior in the literature, the potential for the mapping investigated in this paper to include a large number of relationships was significant, given that we have cataloged 35 code smells and 82 refactorings for Elixir.

To handle this issue, we opted to structure a relational database (RDB) using the SQLite[7] embedded database engine and then use it to persist all the relationships found in the mapping between smells and refactorings. Figure 2 presents the database schema modeled to persist these data. The relationship between the `Smell` and `Refactoring` tables was designed as many-to-many, thus generating the `Smell_Refactoring` table, where all the data found in the subsequent steps of this study were persisted.

According to Bibiano *et al.* [38] and Cedrim *et al.* [39], refactorings are considered incomplete when they fail to remove a code smell completely. Considering that a refactoring might not always have the capability to completely remove an instance of a code smell on its own, we added the `order` column to the `Smell_Refactoring` table. This column is intended to define the position of a refactoring within a sequence of interrelated refactorings aimed at removing the same code smell. In other words, the `order` column allows us to identify composite refactorings [29, 30] for removing code smells in Elixir. When a refactoring is not used in conjunction with other transformation strategies to aid the removal of a smell, this column has a null value persisted in the respective relationship.

The resulting database from the study conducted in this paper, along with a set of useful SQL queries, is available online as part of our replication package[8]. These artifacts support researchers in filtering and reusing subsets of our results in complementary studies and enable developers and tooling teams to build practical tools for automated code

[7] https://www.sqlite.org/

[8] `https://doi.org/10.5281/zenodo.13835771`

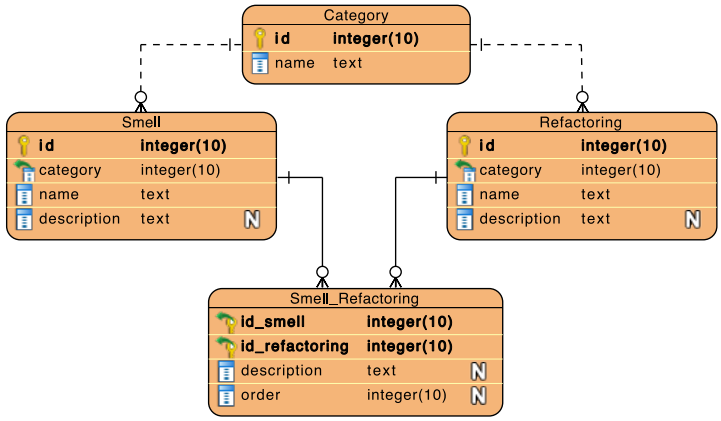

FIGURE 2 Database used to document treatments to code smells by refactoring strategies

smell refactoring in Elixir, for example by extending *RefactorEx* [40], a VS Code plugin for Elixir code refactoring.

2) Mapping Smells to Refactorings: In this step, the first author of this work manually compared each of the 35 code smells for Elixir with the 82 refactorings cataloged for this language. Essentially, the main objective of this step was to establish relationships between code smells and the refactoring strategies capable of contributing to their removal. To achieve this, the characteristics of the problems caused by the code smells were compared with the code generated by each of the Elixir refactorings. When it was identified that a problem caused by a sub-optimal structure is part of the motivation for performing a particular refactoring, which would thus remove it at least partially, a relationship between a code smell and a refactoring was established.

Considering the number of smells and refactorings for Elixir, this step had the potential to identify 2,870 relationships between these two sets, representing their Cartesian product. Out of these, 164 relationships were mapped, involving 35 smells and 59 refactorings. In other words, in this step, it was possible to identify that all the code smells for Elixir have their removal aided by at least one refactoring for this language. However, at the end of this step, 23 refactoring strategies had not yet been associated with the removal of any cataloged code smells for Elixir.

Parallel to this step, the activity *Composite Refactoring Extraction* was also conducted, as shown in Figure 1. In this parallel activity, whenever a relationship between a code smell and a refactoring was identified, the first author of this work empirically analyzed whether the refactoring in question is sufficient to remove the mapped code smell on its own, or if this code transformation needed to be complemented by other refactorings performed in a specific sequence to enhance the removal. This parallel activity therefore allowed us to identify composite refactoring candidates [29, 30] for Elixir, something not directly explored in our previous works on refactorings for this language [21, 41].

3) Mapping Refactorings to Smells: Considering the manual and subjective nature of the mapping carried out in the previous step, and also that a code smell can eventually be removed in different ways depending on the specific problem a developer has at hand [6, 14], in this third methodological step, the first author of this work manually compared each of the 23 refactorings not yet associated with any code smell in the previous step with all 35 code smells cataloged for Elixir. The objective of following the reverse path of the previous step and additionally using only a specific subset of the refactorings was to mitigate any potential flaws in the previous mapping process that may have caused the absence of relationships for these 23 refactorings.

The aspects compared between the code smells and the refactorings in this step were the same as those described

in the step 2, with the main difference being that the initial comparative reference here was the refactorings, not the code smells as previously. Considering the sets of refactorings and code smells compared in this third step, we had the potential to identify 805 relationships between them, representing the Cartesian product of the involved elements. Out of these, 12 new relationships were mapped, involving 11 refactorings and seven smells. Among these refactorings involved in removing code smells, only *Remove import attributes* was mapped in two different removals. The remaining 10 were each mapped to the removal of only one code smell. Regarding the code smells involved in these new relationships, one was mapped four times (*Long Function*) and another three times (*Duplicated code*). The remaining five were each mapped only once. Therefore, at the end of this step, the number of refactoring strategies not associated with the removal of cataloged code smells decreased to 12. In Section 7, we analyze and discuss possible reasons that justify the absence of relationships for these refactorings.

As in the previous step, the *Composite Refactoring Extraction* activity was also conducted parallel to this third step. Thus, after completing the mappings between the code smells and refactorings for Elixir, in addition to identifying how each code smell can be systematically removed with the help of atomic refactoring operations, the first author of this work also identified 26 different interrelated sequences of refactorings that can be used to assist the removal of code smells in Elixir. The resulting mapping and the identified composite refactoring candidates were subsequently reviewed by the second author. This review focused on assessing the plausibility, clarity, and consistency of the proposed associations.

4) Composite Refactorings Analysis: Considering that the *Composite Refactoring Extraction* activity occurred parallel to steps 2 and 3, it was therefore only completed after all the mappings conducted in this work. In total, 28 of the 35 cataloged code smells for Elixir had at least one composite refactoring candidate mapped for their removal.

The objective of this fourth and final methodological step was to quantify how often each of the 26 composite refactoring candidates identified in the previous steps was mapped and to assign names only to those that were recurrent—that is, identical sequences of refactorings applied to the removal of more than one distinct code smell. By naming only recurrent sequences, we avoided proposing compositions that are tightly coupled to a single smell or isolated scenario, ensuring that named composite refactorings reflect patterns of reuse across multiple removal contexts rather than idiosyncratic cases. In total, five of these 26 complex transformations were mapped more than once and were therefore considered composite refactorings for Elixir. Out of these five recurrent composite refactorings, four resemble other complex code transformations previously discussed in the literature in different contexts [6, 29, 42]. Therefore, we used the names of these refactorings as originally proposed by other authors as inspiration for naming these composite refactorings for Elixir.

5 | RESULTS

In Subsection 5.1, we present the mappings identified between Elixir code smells and the corresponding refactorings that can be useful to eliminate them. Moreover, we selected a subset of four code smells to provide a more detailed explanation of how their elimination can be aided by refactoring strategies. These smells were selected for being the most relevant or prevalent in Elixir [20], making their removal important and representative for developers working with this language. In Subsection 5.2, we present five new composite refactorings that contribute to the removal of code smells in Elixir. These composite refactorings were identified during the mappings performed in this study. Finally, in Subsection 5.3, we present a complete example of removing a code smell step-by-step using a composite refactoring in Elixir. For this, all intermediate versions between the original code and the fully refactored code are presented.

5.1 | Mapping between smells and refactorings

In our mapping study, we found that all 35 code smells are covered by at least one of the refactorings in our catalog, meaning there is at least one refactoring that helps in the removal of these smells. In total, 176 relationships between code smells and refactorings for Elixir were identified, demonstrating that a code smell can have their removal aided by more than one distinct refactoring strategy, and also that a refactoring can be useful in the removal of multiple code smells. Due to the large number of relationships between smells and refactorings found in this study, to improve the readability of this paper, we chose to present in Table 1 only the smells that have their removal aided by no more than five different refactorings. Consequently, the mappings for the remaining 10 smells that do not meet this criterion can be found in Appendix A.

Unnecessary macros is the most relevant code smell for Elixir according to the perception of developers who work with this language [20]. This smell occurs whenever a `macro` is used in situations where it would be possible to solve the same problem using functions or other Elixir structures. When a code is implemented as a `macro` but could be implemented as a conventional function in Elixir, we can use the *Inline macro* refactoring in the removal of this smell, thereby replacing all instances of the `macro` with the code defined in its body. When creating a `macro` is unavoidable, but part of it could be implemented as a conventional named function, we can use the refactoring *Extract function* to remove this smell. With this refactoring, we extract part of the macro's code and encapsulate it into a conventional function, which the `macro` will then call. This approach improves code organization and readability while leveraging the macro for its specific role. After extracting the function, it may eventually be necessary to move it to another module to make the code more cohesive. This can be done using the *Moving a definition* refactoring.

Dynamic atom creation is Elixir's second most relevant smell [20]. This smell occurs when a function creates `atoms` in an uncontrolled and dynamic way. Since values of this Elixir basic type are not garbage collected by BEAM, this lack of control by the developer over how many `atoms` will be created during an application's execution cycle can expose the software to unexpected behaviors caused by excessive memory usage. We can contribute to the removal of this smell by replacing calls to the `String.to_atom/1` function, which dynamically creates `atoms`, with explicit conversions. To do this, we should use the *Extract function* refactoring on the calls to `String.to_atom/1` to create a new function. This new function should accept a string as a parameter and convert it to a statically predefined `atom`. The body of this new function should include a conditional that checks the content of the string. Depending on the string's content, the function will return a different predefined `atom` directly, without using `String.to_atom/1`. After extracting a new function for explicit conversions from strings to atoms, we can also use the refactoring *Introduce pattern matching over a parameter* to transform the extracted function into a multi-clause function, where each clause is responsible for returning one of the possible statically predefined atoms.

Instead of extracting new functions to refactor *Dynamic atom creation*, we can also reuse functions implemented previously. If there is already a function in the module responsible for performing the explicit conversion of a string to an atom (*e.g.*, a function extracted for this purpose at a different place in the same module), we can use the refactoring *Folding against a function definition* to replace a call to `String.to_atom/1` with a call to the existing function.

Complex branching is the most prevalent smell among all 35 cataloged for Elixir [20]. This smell occurs when a function takes on the responsibility of handling multiple errors alone, which makes it difficult to maintain and test. When a function uses a conditional statement with many different branches, each responsible for handling a specific error type, we can use the *Extract function* refactoring many times to delegate each branch (handling of a response type) to a different new private function. This approach makes the code cleaner, more concise, and readable. Another possibility to assist the removal of this smell is to use the *Introduce pattern matching over a parameter* refactoring to break down complex branching into a multi-clause function, where each clause handles a different type of error. This

TABLE 1 Elixir smells and the refactorings that assist their elimination

Smell	Refactoring
Shotgun surgery	Moving a definition
Speculative assumptions	Introduce pattern matching over a parameter
	Pipeline using "with"
Complex branching	Extract function
	Introduce pattern matching over a parameter
Using App Configuration for libraries	Add or remove a parameter
	Typing parameters and return values
Large code generation by macros	Extract function
	Moving a definition
Code organization by process	Remove processes
	Remove dead code
Unsupervised process	Moving error-handling mechanisms to supervision trees
	Moving a definition
Feature envy	Extract function
	Moving a definition
	Remove import attributes
Primitive obsession	Grouping parameters in tuple
	From tuple to struct
	Add type declarations and contracts
Unnecessary macros	Inline macro
	Extract function
	Moving a definition
Untested polymorphic behaviors	Introduce overloading
	Folding against a function definition
	Typing parameters and return values
Large messages	Defining a subset of a Map
	Extract expressions
	Add a tag to messages
Large class	Splitting a large module
	Rename an identifier
	Behaviour extraction
	Moving a definition
Divergent change	Splitting a large module
	Moving a definition
	Behaviour extraction
	Rename an identifier
Long parameter list	Add or remove a parameter
	Grouping parameters in tuple
	Reorder parameter
	From tuple to struct
Modules with identical names	Rename an identifier
	Introduce a temporary duplicate definition
	Remove dead code
	Move file

TABLE 1 Elixir smells and the refactorings that assist their elimination (continued)

Smell	Refactoring
Complex else clauses in with	Extract function
	Remove dead code
	Remove redundant last clause in "with"
	Moving "with" clauses without pattern matching
"Use" instead of "import"	Alias expansion
	Remove dead code
	Remove import attributes
	Introduce import
Compile-time global configuration	Extract constant
	Folding against a function definition
	Remove dead code
	Introduce a temporary duplicate definition
Agent obsession	Generalise a function definition
	Add or remove a parameter
	Moving a definition
	Behaviour extraction
GenServer envy	Generalise a process abstraction
	Remove dead code
	Introduce processes
	Register a process
Inappropriate intimacy	Moving a definition
	Closure conversion
	Add or remove a parameter
	Splitting a large module
	Rename an identifier
Dynamic atom creation	Extract function
	Introduce pattern matching over a parameter
	Introduce a temporary duplicate definition
	Remove dead code
	Folding against a function definition
Alternative return types	Introduce a temporary duplicate definition
	Rename an identifier
	Add or remove a parameter
	Typing parameters and return values
	Remove dead code
Using exceptions for control-flow	Rename an identifier
	Introduce a temporary duplicate definition
	Folding against a function definition
	Introduce processes
	Moving error-handling mechanisms to supervision trees

approach enhances readability and maintainability by organizing the code according to distinct error scenarios.

Finally, *Working with invalid data* is the second most prevalent smell for Elixir according to developers [20]. This smell occurs when a function does not validate its parameters and propagates them to other functions, which can lead to unexpected internal behavior. When a library function does not validate the types of its parameters, we can at least create a *Proxy* [43] function for this smelly library function and use the *Typing parameters and return values* refactoring to document the types of these data directly in the proxy. This will help clients of the smelly function (*i.e.*, third-party

code) protect themselves from potential errors caused by invalid data. If recurring data structures are identified while documenting a function using *Typing parameters and return values*, these structures can be named using the *Add Type Declarations and Contracts* refactoring. This approach creates new reusable data types and enhances the system's readability.

The replication package of this paper and Appendix A provide comprehensive details on all the mapped refactoring operations.

> **Finding #1:** All 35 code smells for Elixir have their removal assisted by at least one refactoring also cataloged for this language. Moreover, some of these refactoring operations can be useful for addressing more than one code smell, providing developers with alternatives for solving these issues.

5.2 | Composite refactorings for Elixir

We found five recurring sequences of complementary atomic refactorings that can be useful to remove code smells in Elixir. In other words, in this study, we also cataloged five new composite refactorings for Elixir, as presented in Table 2.

TABLE 2 Composite refactorings for Elixir

Refactoring	Composed by	Related smells
Extract to outside	1. Extract function 2. Moving a definition	Feature Envy Switch statements * Large code generation by macros Data manipulation by migration * Unnecessary macros
Module decomposition	1. Splitting a large module 2. Rename an identifier	Divergent Change Inappropriate Intimacy Large class Data manipulation by migration *
Introduce parameter struct	1. Grouping parameters in tuple 2. From tuple to struct	Long Parameter List Primitive Obsession
Gradual change	1. Introduce a temporary duplicate definition 2. Remove dead code	Dynamic atom creation Modules with identical names
Explicit a changed function signature	1. Add or remove a parameter 2. Typing parameters and return values	Alternative return types * Using App configuration for libraries

* Smell eliminated by composite refactoring used in conjunction with other refactorings.

Extract to outside is a sequence of atomic refactoring operations that contributes to the removal of five different code smells in Elixir, making it the most recurrent composite refactoring among those cataloged in this study. This refactoring is characterized by being a sequence composed of an *Extract function*, followed by a *Moving a definition*. Generally, this sequence can be used to break down an original function into smaller parts and move them to modules that group other functions with similar objectives. Therefore, this composite refactoring aims to improve the understandability of a function while maintaining the cohesion of the module where it is originally defined. The mechanics of *Extract to outside* are analogous to *Method decomposition* [29], which is an equivalent composite refactoring cataloged for object-oriented code.

The second most recurrent composite refactoring for Elixir is *Module decomposition*. Its name is similar to another equivalent composite refactoring—*Class decomposition*—cataloged by Brito *et al.* [29]. Some instances of code smells in Elixir are characterized by modules with too much behavior or modules that collaborate excessively and are too highly coupled. This refactoring can address smells with these characteristics by improving the modularity and cohesion of a system, while also reducing unnecessary coupling between modules. To achieve this, the refactoring employs a sequence composed of *Splitting a large module*, followed by *Rename an identifier*, as breaking a module into smaller ones may require updating the name of the original module to better reflect its new purpose.

Introduce parameter struct is analogous to the traditional refactoring *Introduce parameter object*, cataloged by Fowler [6]. Although this traditional refactoring is not composite, both aim to group related data into a reusable data structure. However, to achieve a result equivalent to *Introduce parameter object* with *Introduce parameter struct*, we need to perform two distinct atomic refactorings sequentially: *Grouping parameters in tuple* and *From tuple to struct*, respectively.

Gradual change is a composite refactoring for Elixir that can occur over several different commits until its completion. It involves code transformations that coexist with the original versions for a certain period, thereby avoiding breaking changes. The expressions or modules involved in this composite refactoring are initially duplicated using the atomic refactoring *Introduce a temporary duplicate definition*. The original versions are set as deprecated while the actual changes are applied to the duplicates. After a period with the original versions deprecated, they are removed using the atomic refactoring *Remove dead code*. The purpose of *Gradual change* is similar to the technique *Branch by Abstraction* [42], commonly used in continuous delivery processes to implement large-scale changes to a software system gradually, allowing for regular releases even while the change is still ongoing.

Finally, *Explicit a changed function signature* is a composite refactoring for Elixir used in situations where, to remove a smell, we need not only to modify a function's signature but also to describe the types of its parameters and return values. This refactoring consists of two atomic refactorings from our catalog applied in sequence. First, *Add or remove a parameter* changes the function's signature, and then *Typing parameters and return values* documents the types for the new signature.

As we can see in Table 2, two of the five composite refactorings cataloged for Elixir in this study—*Introduce parameter struct* and *Gradual change*—have the capability to aid the elimination of the smells they were mapped to without requiring any additional refactorings. The other three composite refactorings sometimes need to be used in conjunction with other refactorings to enhance the removal of the mapped smells. For example, instances of the *Large code generation by macros* smell can be completely removed using just the composite refactoring *Extract to outside*. Similarly, instances of the *Divergent change* smell can be removed using *Module decomposition*. However, during the elimination of *Data manipulation by Migration* instances, we need to apply the following sequence of refactorings: *Module decomposition* → *Extract to outside* → *Remove dead code*[9].

All instances of combined usage of composite refactorings and atomic refactorings mapped for removing code smells in Elixir are described in detail in Appendix A and in the replication package of this paper. To better illustrate how we can remove a code smell in Elixir through a composite refactoring, in the next section, we present a step-by-step example.

> **Finding #2:** We identified five composite refactorings that can be useful in removing smells in Elixir. Three of them can be used in conjunction with other refactorings to aid the removal of the smells they were mapped to.

[9]This sequence is a composite refactoring *candidate* for Elixir. However, since it does not recur in removing other code smells like the five listed sequences in Table 2, it was not named.

As described in Section 4, beyond the five composite refactorings identified in this work, we identified 21 additional refactoring sequences that support code smell removal in Elixir. Because each sequence targets a single specific smell, we did not name them due to their limited reuse potential. These sequences, referred to as *composite refactoring candidates*, are summarized in Table 3 and represent opportunities for future investigation.

TABLE 3 Composite refactoring candidates for Elixir

Composite refactoring candidate	Description	Related smell
Struct field access elimination → Equality guard to pattern matching	Centralizes access to struct fields into a temporary variable and, when applicable, replaces their use with pattern matching in guards, avoiding direct and repeated access to nonexistent fields	Accessing non-existent map/struct field
Generalise a function definition → Moving a definition → Add or remove a parameter	Centralizes access to the `Agent` in a generic function, relocates this responsibility to another module, and adjusts the required parameters, reducing code dispersion and coupling to the `Agent`	Agent Obsession
Introduce a temporary duplicate definition → Rename an identifier → Explicit a changed function signature* → Remove dead code	Splits the original function into specialized functions for each return type, renames them, and makes their signatures explicit, removes conditional parameters and dead code, and makes return types clear and predictable	Alternative return types
Remove processes → Remove dead code	Replaces processes used solely for organizational purposes with conventional modules and removes obsolete callbacks and functions, simplifying the code and improving comprehensibility	Code organization by process
Folding against a function definition → Remove dead code	Replaces compile-time constants with `Application Environment` calls and removes obsolete global definitions, making configuration more flexible and better suited to runtime	Compile-time global configuration
Extract function → Remove dead code	Extracts error handling into dedicated functions and removes the `else` clause from the `with` statement, simplifying control flow and improving code clarity and readability	Complex else clauses in with
Introduce a temporary duplicate definition → Temporary variable elimination	Moves complex extractions from the clause into the function body using temporary variables and then eliminates unnecessary ones, simplifying clauses and improving code readability	Complex extractions in clauses
Module decomposition* → Extract to outisde* → Remove dead code	Separates data and schema migrations into distinct modules, moves data manipulation logic into a dedicated module, and removes unnecessary code, making migrations more cohesive	Data manipulation by migration
Introduce Enum.map/2 → Transform to list comprehension → List comprehension simplifications	Replaces manual and repeated generation of list elements with abstractions such as `Enum.map/2` or list comprehensions, eliminating duplication and making the code more idiomatic	Duplicated code
Extract function → Introduce pattern matching over a parameter	Replaces dynamic `atom` creation with a multi-clause function using pattern matching to explicitly map `strings` to known `atoms`, avoiding unsafe `atom` generation at runtime	Dynamic atom creation
Generalise a process abstraction → Introduce processes → Register a process → Remove dead code	Refactors `Agents` or `Tasks` that assume excessive responsibilities into more appropriate `GenServers`, introduces and registers processes as required by concurrency needs, and removes obsolete code, better aligning processes with the problem domain	GenServer envy
Closure conversion → Add or remove a parameter	Transforms closures into pure anonymous functions by making dependencies explicit through parameters and adjusting their signatures, reducing implicit coupling between scopes	Inappropriate Intimacy

* Composite refactoring within a composite refactoring candidate.

TABLE 3 Composite refactoring candidates for Elixir (continued)

Composite refactoring candidate	Description	Related smell
Convert nested conditional to pipeline → Replace function call with raw value in pipeline start	Replaces nested conditionals with functional pipelines and simplifies the beginning of the pipelines, reducing function size and making the code more idiomatic and readable	Long Function
Replace conditional with polymorphism via Protocols → Convert guards to conditionals	Replaces chains of conditionals with polymorphism via `Protocols` and combines this approach with guards, making control flow more extensible and reducing duplication	Switch statements
Extract to outside* → Generalise a function definition → Folding against a function definition	Isolates the conditionals into a dedicated function, generalizes it through a strategy passed as a parameter, and replaces duplicated switches with reusable calls, thus improving extensibility	Switch statements
Rename an identifier → Function clauses to/from case clauses	Regroups clauses with related responsibilities through renaming to form distinct functions and optionally converts multi-clause definitions into `case` expressions	Unrelated multi-clause function
Introduce overloading → Folding against a function definition	Makes polymorphism explicit through multi-clause functions, one per supported type, and reuses logic across clauses, making each case clearer, more isolated, and easier to test	Untested polymorphic behaviors
Introduce import → Remove dead code	Replaces the improper utilization of `use` with `import` or `alias`, removing the unnecessary directive and avoiding the implicit propagation of dependencies between modules	"Use" instead of "import"
Rename an identifier → Introduce a temporary duplicate definition → Folding against a function definition	Renames the function that raises exceptions to the `!`-suffixed variant, introduces a non-raising version without `!` that returns result tuples `{:ok, _}` or `{:error, _}` instead of exceptions, and implements the `!` version on top of the non-raising one	Using exceptions for control-flow
Introduce processes → Moving error-handling mechanisms to supervision trees	Transforms the module containing the exception-raising function into a process and moves error handling to the supervision tree, adopting the *"let it crash"* style	Using exceptions for control-flow
Typing parameters and return values → Add type declarations and contracts	Explicitly documents parameter and return types and defines reusable types for recurring structures, reducing the use of invalid data and facilitating validation by API clients	Working with invalid data

* Composite refactoring within a composite refactoring candidate.

5.3 | Example: Removing a smell step-by-step through a composite refactoring

In this section, the smell *Large code generation by macros* is removed step-by-step using the composite refactoring *Extract to outside* to illustrate how this type of refactoring can be applied in Elixir. As shown in Table 2, this composite refactoring is characterized by the following sequence of atomic refactorings: *Extract function* → *Moving a definition*.

Macros are meta-programming mechanisms in Elixir that can extend the language[10]. They enable robust code generation at compile time, which helps reduce boilerplate and allows the creation of DSL constructs for Elixir [2]. 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 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 2 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).

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

LISTING 2 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 3, 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 3 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
```

The first step in removing this smell is to perform *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 4, by extracting to the function `__define__/3` (line 9) the code originally defined inside the `quote/1` (Listing 3 - 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 4 Intermediate code version after *Extract function*

```
1  defmodule Routes do
2    ...
3    defmacro get(route, handler) do
```

```
    quote do
      Routes.__define__(__MODULE__, unquote(route), unquote(handler))
    end
  end

  def __define__(module, route, handler) do
    if not is_binary(route) do
      raise ArgumentError, "route must be a binary"
    end

    if not is_atom(handler) do
      raise ArgumentError, "handler must be a module"
    end

    Module.register_attribute(module, :store_routes, accumulate: true)
    Module.put_attribute(module, :store_routes, {route, handler})
  end
end
```

After extracting the function `__define__/3` into the `Routes` module (Listing 4), we can also move it to the `Routes.Utils` module using the *Moving a definition* refactoring, thus making the code more cohesive (Listing 5). This occurs because, although omitted in Listing 5, `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 5 Code smell is completely removed after using a composite refactoring

```
defmodule Routes do
  ...
  defmacro get(route, handler) do
    quote do
      Routes.Utils.__define__(__MODULE__, unquote(route), unquote(handler))
    end
  end
end
```

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

    if not is_atom(handler) do
      raise ArgumentError, "handler must be a module"
    end

    Module.register_attribute(module, :store_routes, accumulate: true)
    Module.put_attribute(module, :store_routes, {route, handler})
  end
end
```

As introduced earlier, the Phoenix Framework is one of the primary technologies for software development in

Elixir. According to the Stack Overflow Developer Survey[11], it has been the most admired web framework since 2023, with 79% of more than 49k respondents reporting this rating in 2025.

Phoenix is also highly popular on GitHub, with approximately 23k stars, ranking fourth most starred among about 123k Elixir repositories. The refactoring of the *Large code generation by macros* smell presented in this section is grounded in a concrete real-world case from the Phoenix codebase[12], specifically the `Phoenix.Router` module, which makes extensive use of macros. Because these macros are expanded repeatedly, they resulted in large scopes that negatively affected compilation time. To mitigate this issue, the developers defined functions once and invoked them repeatedly, leading to a substantial reduction in compilation time.

6 | VALIDATION

In this section, we report the details of a survey conducted with developers who work with the Elixir programming language, aiming to validate the relationships between code smells and refactorings for this language previously presented in Section 5 and detailed in Appendix A. More specifically, we sought to quantify developers' perceptions regarding the degree of completeness with which each of the 35 code smells cataloged for Elixir can be removed through the sets and sequences of refactorings mapped to support this code quality improvement process.

To this end, Subsection 6.1 presents the methodological steps adopted to design the survey, recruit the participating developers, and analyze the collected data. Finally, Subsection 6.2 presents developers' perceptions of how completely each Elixir code smell can be removed through the mapped refactorings.

6.1 | Survey design

Although the code smells and refactorings considered in this study had been previously validated by more than 300 developers through multiple surveys in earlier works [20, 21], we decided to conduct a new survey in the present study because the relationship between these constructs had not yet been examined from the developers' perspective in prior research, thus requiring explicit validation. Figure 3 summarizes the steps used to conduct this survey, which clarifies how Elixir developers perceive the completeness of code smell removal achieved through refactoring strategies. The steps are detailed in the following paragraphs.

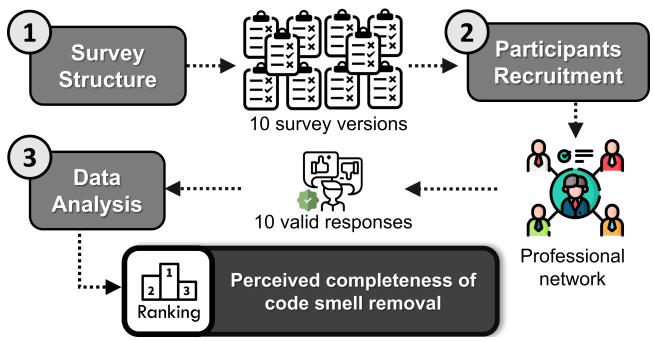

FIGURE 3 Overview of survey on code smell removal in Elixir

[11] https://survey.stackoverflow.co/2025/

[12] https://github.com/phoenixframework/phoenix/blob/675e924f73da1a594e07ba22fcbda6636790ce4a/lib/phoenix/router.ex#L323

1) Survey Structure: The data collection instrument opened with contextual information about the study, including its purpose, the researchers involved, and the voluntary character of participation. Participants were explicitly informed about confidentiality and anonymity assurances and were required to formally agree to an informed consent statement prior to accessing the questionnaire. The complete set of survey items is provided in the replication package accompanying this study. The questions were then arranged according to the following thematic categories:

- *Demographics*: Initially, we gathered demographic and professional background data, including participants' geographic region, their accumulated experience with the Elixir language, and the extent of their involvement in Elixir-based projects.
- *Perceptions on code smell removal using refactoring strategies in Elixir*: Subsequently, participants were presented with a set of Elixir code smells together with the corresponding sequences of refactoring strategies that had been mapped to support their removal. Each code smell was introduced with a brief explanation, followed by descriptions of how each associated refactoring could contribute to its removal, in a manner consistent with the practical guidelines presented in Appendix A. For each smell, we then asked participants whether they agreed that the proposed sequence of refactorings effectively removes the smell. Responses were provided using a five-point Likert scale ranging from one (*the refactoring sequence does not help remove the smell at all*) to five (*the refactoring sequence completely removes the smell*). For every rating, participants were also given the opportunity to provide a justification for their assessment, allowing them to add contextual information and thereby enrich their responses.
- *Final remarks*: At the conclusion of the questionnaire, participants were invited to provide open-ended feedback regarding the study and the data collection instrument.

Given that asking participants to assess all 35 code smell removal strategies would be time-consuming and potentially fatiguing, since it would require validating each of the 176 code smell–refactoring relationships identified in this study, we designed ten distinct versions of the survey. Each version elicited participants' perceptions regarding the removal of exactly seven code smells. As a result, each of the 35 code smells appeared in two different survey versions, enabling us to collect two independent evaluations per smell from different developers. The assignment of code smells to survey versions was randomized. This process resulted in an acceptable range of 30 to 39 relationships per survey version.

A separate Google Form was created for each survey version. The complete distribution of code smells across survey versions, as well as the full versions of the questionnaires used in this study, are available in the replication package.

2) Participants Recruitment: Although we sought to minimize the time and effort required to complete the survey, we acknowledge that this task is inherently time-consuming due to the large number of relationships mapped in the present study. For this reason, we opted to recruit ten Elixir developers directly from our professional networks rather than conducting an open call to the broader developer community through official language channels, as done in our previous studies [20, 21]. We considered this approach more suitable for increasing the likelihood of engaging participants willing to commit to the required effort, as well as reducing the risk of partial or abandoned responses.

The entire recruitment process and data collection phase lasted one month. Each of the ten recruited participants completed a different version of the questionnaire, with the assignment of versions to participants being randomized. The participant pool consisted of seven men and three women, all employed by seven different companies that use Elixir in production environments. Seven participants were based in Brazil, while one participant each was based in Germany, Italy, and Poland. All participants had more than three years of experience with Elixir, and nine of them had

worked on more than four distinct projects using the language. These characteristics highlight both the diversity and the relevance of the participants involved in validating the findings of this study.

Nevertheless, we acknowledge that the relatively small number of participants constitutes a limitation of this validation survey. Therefore, the results should be interpreted as exploratory evidence rather than conclusions that can be generalized to the entire developer community. This design reflects a trade-off between breadth and depth, as recruiting a smaller group of experienced practitioners and assigning each participant a different questionnaire version allowed us to obtain detailed evaluations for a large set of relationships that would likely be impractical to assess within a single survey instrument. Future studies could extend this validation with a larger and more diverse sample of participants.

3) Data Analysis: To analyze the collected responses, we employed descriptive statistics. As described earlier, the perceived completeness of removal for each of the 35 Elixir code smells was independently assessed by two developers. To obtain a single representative value per smell, we normalized these assessments by computing the arithmetic mean of the two ratings. Higher mean values, particularly those closer to five, indicate that developers perceived the corresponding code smell removal as more comprehensive or complete, suggesting that the refactoring sequences mapped to support its removal effectively fulfill their intended purpose.

In addition to the arithmetic mean, we computed the percentage of exact agreement between the two evaluators as a direct and interpretable indicator of response consistency and inter-rater reliability [44]. To further characterize the degree of disagreement, we calculated the mean absolute difference (MD), which quantifies, on average, the number of scale levels separating the two evaluators' ratings for each code smell. Distance-based measures such as MD are particularly suitable for ordinal scales, especially when the number of raters is small, as in the context of our survey [45].

Finally, we analyzed the frequency of high-divergence cases, defined as instances in which the two ratings for the same code smell removal differed by at least two scale levels. This analysis follows prior recommendations to explicitly identify substantive disagreements in subjective assessments [46]. Such cases exceed the minor variations commonly expected in ordinal judgments and may reflect ambiguities in code smell definitions, challenges in evaluating their removal, or strong context dependence. Taken together, the analysis of high-divergence cases complements the other metrics by highlighting evaluations that may be more problematic from an interpretative perspective.

6.2 | How do developers perceive the removal of code smells in Elixir through refactorings?

To provide an overview of developers' perceptions regarding the completeness of code smell removal in Elixir, we first analyze the aggregated distribution of the collected ratings. For each code smell, the reported completeness level corresponds to the arithmetic mean of two independent evaluations, measured on an ordinal scale ranging from 1 (*the refactoring sequence does not help remove the smell at all*) to 5 (*the refactoring sequence completely removes the smell*). As illustrated in Figure 4, a strong concentration of ratings can be observed at the upper end of the scale, with 48.6% of the assessments at level 5 and 34.3% at level 4, totaling 82.9% of the ratings in the two highest levels. These results indicate that, overall, the refactoring strategies proposed for Elixir were perceived by developers as achieving largely or fully complete code smell removal. In contrast, the lower levels of the scale (1 to 3) account for only 17.2% of the ratings, suggesting that partial or incomplete removals are less frequent, although still present.

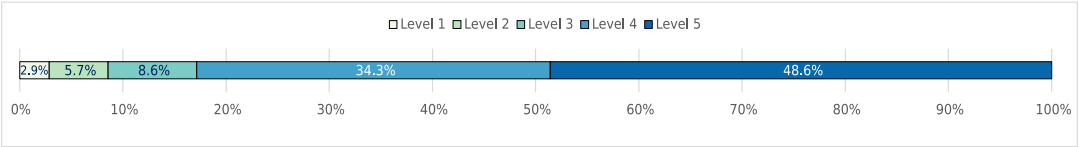

FIGURE 4 Aggregated distribution of removal completeness levels for Elixir code smells

Finding #3: Most Elixir code smells received high removal completeness ratings, with 48.6% of the assessments at level 5 and 34.3% at level 4, together representing 82.9% of all evaluations. This pronounced concentration at the upper end of the scale provides strong empirical evidence that the mapped refactoring sequences effectively support the removal of the majority of Elixir code smells.

The removal completeness levels for each of the 35 Elixir code smells are shown individually in Figure 5, where smells are ordered in descending order according to their perceived completeness of removal. In addition to the mean value, each bar is accompanied by a blue error bar representing the variation between the two ratings received for each smell. For instance, the *Agent Obsession* code smell achieved a removal completeness level of 5, as both evaluators assigned the same maximum rating. In contrast, the *Working with invalid data* smell obtained a removal completeness level of 4, reflecting ratings of 5 and 3 from its respective evaluators.

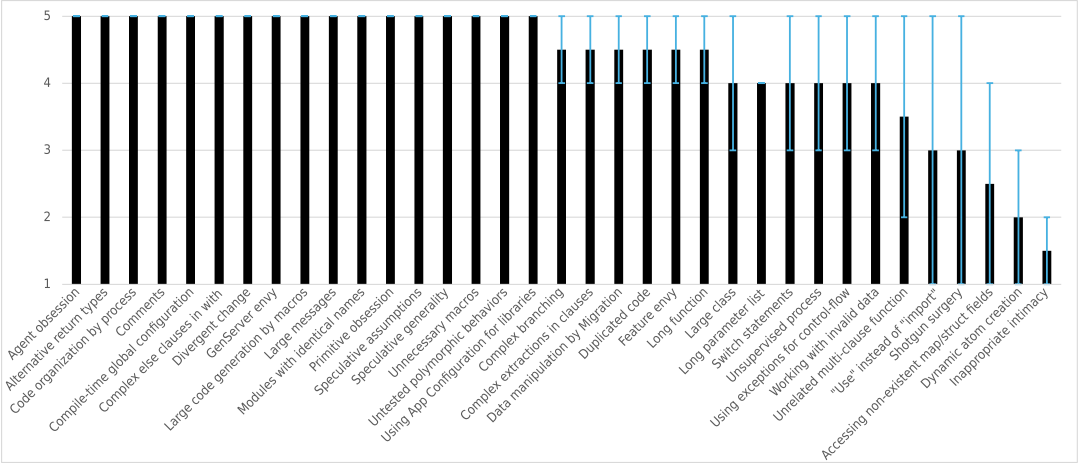

FIGURE 5 Developers' perceptions of the completeness of code smell removal in Elixir using refactorings

Inter-rater agreement was analyzed using complementary metrics aimed at capturing both exact agreement and the magnitude of observed disagreements. The mean absolute difference (MD) was 0.94, indicating that, on average, the ratings assigned to the completeness of code smell removal differed by approximately one scale level. This value suggests a moderate level of disagreement, which is expected in assessments of a perceptual nature.

Regarding exact agreement, 18 out of the 35 code smells (51.43%) received identical ratings from both evaluators, as illustrated in Figure 5. This proportion indicates a reasonable degree of consensus, particularly given the five-point scale granularity and the inherent subjectivity involved in assessing code smell removal completeness.

An analysis of the divergence distribution revealed that 10 out of the 35 smells (28.57%) exhibited high divergence between evaluators, defined as a difference of at least two scale levels. Among these, four smells (11.43%) showed very high divergence (differences of three levels or more), and only two smells (5.71%) exhibited extremely high divergence (differences of four levels). These results indicate that, although instances of substantial disagreement exist, most divergences are moderate, with only a few cases of severe disagreement.

The predominance of ratings at levels 4 and 5, particularly evident in Figure 4, reinforces the findings of the inter-rater agreement analysis, especially the low mean absolute difference (MD = 0.94). Together, these results indicate that, in addition to a generally positive assessment, there was also relatively close alignment between the evaluators' ratings.

> **Finding #4:** The evaluations demonstrate high inter-rater consistency. Exact agreement was observed in 51.43% of the assessments, while the mean absolute difference was low (0.94), indicating that, on average, the ratings for the removal completeness of each smell differed by no more than one scale level. Very high-divergence cases were rare, accounting for only 11.43% of the evaluated removals. Taken together, these results provide strong empirical support for the reliability of the reported findings.

Two cases stand out with respect to the extreme divergences observed: the *"Use" instead of "Import"* and *Shotgun Surgery* code smells. Both received ratings of 1 and 5 from their respective evaluators, corresponding to the largest divergences identified in the study. In the case of *"Use" instead of "Import"*, the evaluator who assigned a rating of 1 justified the low score by arguing that the smell was not relevant within the specific context of their codebase:

> *"In my projects, I generally employ "`use`" to create abstractions that can be used by many `modules`, both bringing aliases and functions. So I don't think it's a strong smell [...]"*

Although this participant's evaluation may have been influenced by their perception of the smell's relevance, the *"Use" instead of "Import"* smell was classified as moderately relevant in a prior study involving 181 Elixir developers [20].

Similarly, for the *Shotgun Surgery* smell—which refers to situations in which a single conceptual change in the code requires many small, coordinated modifications scattered across multiple modules during maintenance and software evolution activities [6]—the evaluator's justification suggests a possible misunderstanding of the smell's definition:

> *"The smell definition shows occurences [...] in a single module, so condensating things into a module doesn't remove the smell."*

The removal strategy proposed for *Shotgun Surgery* in Elixir is primarily based on repeated applications of the *Moving a definition* refactoring, which closely mirrors the traditional approach to addressing this smell as originally described by Fowler [6]. This further supports the interpretation that the extremely low rating assigned to this smell removal constitutes an outlier responsible for the observed high divergence.

Finally, only the *Inappropriate intimacy* code smell exhibited consistently low removal completeness ratings. In this case, the results indicate that the refactoring strategies currently mapped to its removal remain insufficient, highlighting the need for future investigations into more effective approaches for addressing this smell in Elixir.

7 | DISCUSSION

In this section, we present and discuss key insights derived from this study. Specifically, Subsection 7.1 analyzes the refactorings that were not associated with code smell removal and discusses possible reasons for their absence. Finally, Subsection 7.2 explores the potential generalization of our findings to functional programming languages beyond Elixir.

7.1 | Insights from unmapped refactorings

Since 12 of the 82 refactorings cataloged for Elixir were not mapped to removing any code smell, we sought to understand the reasons for these mapping absences. To do this, we used a taxonomy proposed by Abid *et al.* [19] to classify the main motivations behind performing a refactoring strategy. After the first author of this paper classified the motivations behind each of the 12 code transformation strategies not mapped to the removal of smells, the second author validated these classifications, which are presented in Table 4.

TABLE 4 Refactorings not mapped to the removal of code smells

Refactoring	Motivation
From meta to normal function application	Internal quality
Group case branches	Performance
Improving list appending performance	Performance
Nested list functions to comprehension	Performance
Remove single pipe	Internal quality
Replace "Enum" collections with "Stream"	Performance
Replace a nested conditional in a "case" statement with guards	Internal quality
Replacing recursion with a higher-level construct	Internal quality
Transform "if" statements using pattern matching into a "case"	Internal quality
Transform "unless" with negated conditions into "if"	Internal quality
Transform a body-recursive function to a tail-recursive	Performance
Transforming list appends and subtracts	Internal quality

According to the taxonomy used in this classification, the motivation for improving internal quality involves, for example, maintainability, flexibility, portability, reusability, or readability code issues. On the other hand, the motivation for improving performance involves aspects such as response time, error rate, request rate, memory use, or code parallelization. In addition to these two motivations, the taxonomy proposed by Abid *et al.* [19] includes three other categories which clearly do not have any relation with the 12 unmapped refactorings: external quality, security, and migration issues.

As shown in Table 4, five of the 12 unmapped refactorings are motivated by performance concerns. This explains their absence in removing code smells in Elixir, as these sub-optimal code structures are more related to aspects that hinder the internal quality of systems [6]. On the other hand, the other seven unmapped Elixir refactorings focus on improving internal quality, which at first could suggest the existence of uncataloged code smells for Elixir. However, six out of these seven refactorings primarily address minor coding style adjustments, such as replacing certain conditional statements or substituting calls to Elixir's built-in functions with specific operators of this language. Given that code

smells generally involve structures with more substantial granularities, this explains the absence of mappings for these six refactorings. The lack of mappings for *Replacing recursion with a higher-level construct*, however, might indeed indicate the existence of an uncataloged code smell. This refactoring involves significant code block transformations, potentially impacting the design of functions or modules. This type of change is compatible, for example, with the removal of an uncatalogued *Design-related* smell, as described in our catalog [20].

Since Elixir does not have classical iteration constructs (*e.g.*, `while` and `do..while`), recursion is the primary looping mechanism used in this language. However, given that Elixir also provides many higher-order functions that enable iteration while hiding the details of recursion (*e.g.*, `Enum.map/2` and `Enum.reduce/3`), using explicit recursion might be considered a code smell when a built-in higher-order function could be used instead. This is because *Unnecessary explicit recursion*, as we refer to this new smell, can make the code verbose and harm its understandability, thus requiring developers to use greater cognitive load to grasp their purposes, especially when the code is developed by someone else. With the help of the refactoring *Replacing recursion with a higher-level construct*, which initially was not mapped to the removal of any code smell, we can transform the body of recursive functions into calls to higher-order functions, making the code more concise, easier to understand, and consequently easier to maintain.

Listing 6 presents two distinct instances of the *Unnecessary explicit recursion* code smell. The multi-clause functions `factorial/1` (lines 2 and 3) and `sum_list/1` (lines 5 to 8) explicitly use recursion as an iteration mechanism, even though these implementations could instead rely on higher-order functions provided by Elixir.

LISTING 6 Instances of the previously uncatalogued *Unnecessary Explicit Recursion* smell

```
1 defmodule Foo do
2   def factorial(0), do: 1
3   def factorial(n), do: n * factorial(n - 1)
4
5   def sum_list([]), do: 0
6   def sum_list([head | tail]) do
7     head + sum_list(tail)
8   end
9 end
```

The removal of the two instances of the *Unnecessary explicit recursion* smell shown in Listing 6 can be achieved through the *Replacing recursion with a higher-level construct* refactoring, as illustrated in Listing 7. After refactoring, the `factorial/1` (lines 2 to 4) and `sum_list/1` (lines 6 to 8) functions are no longer recursive and instead invoke the built-in higher-order function `Enum.reduce/3`. This function takes as parameters a list to be iterated, an initial accumulator value, and an anonymous function. The anonymous function is invoked to each list element together with the accumulator, and its return value is used as the accumulator for the next iteration. The final accumulator is returned as the result, which corresponds to the values returned by `factorial/1` and `sum_list/1` after refactoring.

LISTING 7 Complete removal of the *Unnecessary Explicit Recursion* smell through refactoring

```
1 defmodule Foo do
2   def factorial(n) do
3     Enum.reduce(1..n, 1, &(&1 * &2))
4   end
5
6   def sum_list(list) do
7     Enum.reduce(list, 0, &(&1 + &2))
8   end
9 end
```

Finding #5: We have found evidence suggesting the existence of one uncatalogued *Design-related* smell for Elixir.

Regarding the unmapped refactorings, as shown in Table 5, the *Traditional* refactorings category [21] had the lowest proportion of refactoring strategies not associated with code smell removal in Elixir (8.00%). Furthermore, 51.14% of the 176 relationships mapped in this study involved *Traditional* refactorings, making it the category that contributes the most to code smell removal. These data show that refactorings cataloged 25 years ago by Fowler and Beck [6], although originally proposed for a different context, remain highly useful and relevant even for a specific scenario like a functional language such as Elixir.

TABLE 5 Overview of refactorings by category (Unmapped x Mapped)

Category	# Refactorings	# Unmapped (%)	# Relationships (%)
Traditional	25	***2 (8.00)***	***90 (51.14)***
Functional	32	7 (21.88)	46 (26.14)
Erlang-Specific	11	1 (9.09)	21 (11.93)
Elixir-Specific	14	2 (14.29)	19 (10.80)

Finding #6: *Traditional* refactorings proposed to improve the quality of object-oriented systems are also highly important for removing code smells in Elixir.

7.2 | Beyond Elixir

As discussed in Section 3, prior studies have investigated the existence of code smells and refactorings specific to individual functional languages, such as Erlang and Haskell. However, to the best of our knowledge, this is the first study to explicitly correlate catalogs of code smells and refactorings within the context of a functional programming language. Because our investigation is grounded in Elixir, and given the variability in constraints and refactoring challenges across programming languages [8], we cannot claim that all findings reported here are directly transferable to other functional languages.

In this context, future work should replicate and extend our approach by cataloging and correlating code smells and refactorings in other functional languages, such as Clojure, F#, Scala, Julia, and related ecosystems. Once such language-specific studies are available, comparative analyses of their results may enable the identification of code smells and refactorings that are characteristic of the functional paradigm as a whole. This line of research could ultimately support a level of generalization for functional programming analogous to that achieved by Fowler and Beck [6] for the object-oriented paradigm.

Although generalization beyond Elixir was not an explicit goal of this present study, we speculate that the removal of *Traditional* code smells (see Table 6), as defined in the Elixir smells catalog [20], when performed through refactorings classified as *Traditional* or *Functional* (see Table 7) in the corresponding Elixir refactoring catalog [21], may also be applicable to other functional languages. This speculation is grounded in the fact that *Traditional* smells and refactorings are generally more language-agnostic, while *Functional* refactorings for Elixir rely on paradigm-level concepts such as pattern matching, higher-order functions, pipelines, list comprehensions, and declarative data transformations, which are commonly supported across functional languages.

TABLE 6 Code smells potentially applicable beyond Elixir

Traditional code smells			
Comments (A.5)	Divergent Change (A.11)	Duplicated Code (A.12)	Feature Envy (A.14)
Inappropriate Intimacy (A.16)	Large Class (A.17)	Long Function (A.20)	Long Parameter List (A.21)
Primitive Obsession (A.23)	Shotgun Surgery (A.24)	Speculative Generality (A.26)	Switch Statements (A.27)

(A.x) Corresponding removal strategy listed in Appendix A.

More broadly, our findings provide indications that some insights may extend to other functional languages. First, traditional design problems (*e.g.*, overly complex or poorly modularized code) may also arise in functional systems and can often be addressed using well-established refactoring strategies. Second, functional-language constructs such as pattern matching and higher-order functions may enable concise refactoring solutions for restructuring code and reducing duplication. Third, organizing refactorings around paradigm-level concepts, rather than language-specific syntax, could represent a useful direction for future research aimed at structuring catalogs that support cross-language reasoning within the functional programming ecosystem.

However, confirming these possibilities requires systematic, language-specific empirical studies adopting a methodological design similar to that used in this study.

TABLE 7 Refactorings potentially applicable beyond Elixir

Traditional refactorings *		
Add or remove a parameter	Extract constant	Extract expressions
Extract function	Folding against a function definition	Grouping parameters in tuple
Inline function	Introduce a temporary duplicate definition	Introduce import
Introduce overloading	Move expression out of case	Move file
Moving a definition	Reduce a boolean equality expression	Remove dead code
Remove import attributes	Remove nested conditional statements in function calls	Rename an identifier
Reorder parameter	Replace conditional with polymorphism via Protocols	Simplifying checks by using truthness condition
Splitting a large module	Temporary variable elimination	
**Functional refactorings ** **		
Closure conversion	Converts guards to conditionals	Convert nested conditionals to pipeline
Eliminate single branch	Equality guard to pattern matching	From tuple to struct
Function clauses to/from case clauses	Generalise a function definition	Inline macro
Introduce Enum.map/2	Introduce pattern matching over a parameter	List comprehension simplifications
Merging match expressions into a list pattern	Merging multiple definitions	Replace function call with raw value in a pipeline start
Replace pipeline with a function	Replacing recursion with a higher-level construct	Simplifying guard sequences
Simplifying pattern matching with nested structs	Splitting a definition	Static structure reuse
Struct field access elimination	Struct guard to matching	Transform to list comprehension
Turning anonymous into local functions	Widen or narrow definition scope	

* Only the 23 refactorings that were mapped to code smells.

** Only the 26 refactorings that were mapped to code smells.

8 | THREATS TO VALIDITY

Construct Validity: Considering that a single refactoring might not always be sufficient to fully eliminate a code smell, a threat to the construct validity of this study is that our mapping might lead developers to perform incomplete refactorings [38, 39] due to a lack of knowledge about the necessary complementary steps to completely remove a smell, or at least address the largest possible portion of it. To mitigate this threat, we not only mapped simple relationships between smells and atomic refactorings, but also documented 26 composite refactoring candidates and five composite refactorings for Elixir. These sequences of code transformations used together can help developers reduce the occurrence of incomplete refactorings in Elixir codebases, thereby promoting smell removals that are more aligned with real-world needs. Regarding the construct validity of the survey conducted in this study, a primary concern lies in how participants interpret the concept of removal completeness. Although this construct was operationalized using a five-point Likert scale, developers may have held different understandings of what constitutes a complete removal, particularly for code smells that are more abstract or highly context dependent. To mitigate this threat, each code smell was presented alongside a clear definition and a detailed explanation of the associated refactoring sequence, all derived from previously validated catalogs. In addition, the scale anchors were explicitly specified, and participants were encouraged to provide written justifications for their ratings. These justifications allowed us to better contextualize individual responses and to identify potential misunderstandings. Finally, while aggregating the two ratings per smell using the arithmetic mean may raise concerns due to the ordinal nature of the scale, we mitigated this limitation by complementing the analysis with additional measures that explicitly capture both agreement and disagreement between evaluators.

Conclusion Validity: The primary concern regarding this type of threat in our mapping study relates to potential biases in qualitative analyses that could compromise the reliability of the work. Since all the initial mappings between smells and refactorings, as well as the identification of composite refactoring candidates, were conducted solely by the first author of this study, these analyses might have been influenced by personal experiences and perspectives, potentially compromising the results. To address this threat, the resulting mappings and the identified composite refactoring candidates were also discussed with the second author of this work, who reviewed, agreed with, and validated the decisions made by the first author. In addition, all 176 code smell–refactoring relationships identified in this study were validated through a survey involving ten experienced Elixir developers. This validation allowed us to confirm that the vast majority of Elixir code smells (83%) are perceived as being either fully or largely removable through the proposed refactoring strategies. With respect to the conclusion validity of the survey, a potential threat arises from the limited number of evaluators per code smell, as each smell was assessed by only two developers. This constraint may reduce statistical robustness and increase sensitivity to individual judgments or outliers. To mitigate this threat, we deliberately relied on descriptive and distance-based measures that are well suited to ordinal data and small numbers of raters, including exact agreement, mean absolute difference (MD), and the frequency of high-divergence cases. Rather than drawing conclusions solely from aggregated averages, we explicitly examined and discussed instances of substantial disagreement, thereby reducing the risk of misleading interpretations based on summary statistics alone.

Internal Validity: The main threat to the internal validity of this study concerns the possible existence of relationships between smells and refactorings not captured by the methodological steps employed, which could influence the quality of our findings. Given the manual and subjective nature of the mapping process carried out in this study, it is natural to consider that it is susceptible to human error in identifying these relationships. To mitigate this risk, in addition to conducting an initial comparison between all 35 smells and 82 refactorings for Elixir (Section 4 - step 2), we also implemented a second comparison activity (Section 4 - step 3) aimed at finding relationships for the 23

refactorings not associated with any code smell in the previous step. This additional step helped to reduce potential flaws in this manual comparison process, as it allowed us to find relationships for 11 of the 23 refactorings initially not correlated to the removal of code smells. Additionally, we used the taxonomy proposed by Abid *et al.* [19] to classify the motivations behind the 12 refactorings not associated with code smells and concluded that this absence of relationships is not related to the methods used in this study. Concerning the internal validity of the survey, participant fatigue and cognitive overload constitute potential threats, given the substantial number of code smell–refactoring relationships evaluated in this study. To mitigate this issue, we designed ten distinct versions of the survey, each including only seven code smells and at most 39 relationships out of the 176 mapped in total. In addition, participants were instructed to evaluate no more than one code smell per day. This strategy, combined with Google Forms' support for saving partial responses, helped reduce time pressure and preserve the quality of the evaluations. Another potential threat to internal validity arises from the influence of participants' individual professional contexts and prior experiences on their judgments. We addressed this concern by recruiting developers from different companies, countries, genders, and application domains, thereby reducing the impact of any single contextual perspective. Finally, in a small number of cases, the qualitative justifications indicated that some participants may have conflated the perceived relevance of a code smell with the completeness of its removal. Such cases were identifiable through the open-ended responses and were explicitly acknowledged and discussed in the analysis.

External Validity: This threat concerns the generalization of the relationships found between the catalogs of code smells and refactorings, since these smells may not represent all possible quality issues in Elixir systems, and these refactorings might not be the only ways to address these sub-optimal structures. Despite this risk, both catalogs compared in this study have been extensively validated with experienced developers in previous studies [20, 21]. Additionally, the mappings between both catalogs include refactorings that are useful for removing all cataloged smells, and among the refactorings not associated with smell removal, only one shows indications of an uncataloged smell for Elixir. This suggests the robustness and high level of completeness of our results, which mitigates this threat. Finally, the external validity of the survey conducted in this study is subject to limitations related to both sample size and recruitment strategy. The study involved ten participants, all recruited through the authors' professional networks, which may introduce selection bias and constrain the generalizability of the findings. In particular, the relatively small number of participants should be interpreted as providing exploratory evidence rather than definitive conclusions about the broader Elixir developer community. This recruitment approach was nevertheless adopted deliberately, given the inevitably long time required to complete each questionnaire version, to maximize participant commitment and minimize incomplete or abandoned responses. Moreover, the survey design required distributing different questionnaire versions across participants so that a large set of relationships could be evaluated, which would likely be impractical to assess within a single survey instrument. Although the sample size is limited, all participants were highly experienced Elixir developers, and the core constructs assessed in this study had already been validated by more than 300 developers in prior work [20, 21]. Consequently, while the results may not generalize to novice developers or to programming languages beyond Elixir, they provide credible and relevant insights into how experienced practitioners perceive the effectiveness of refactoring strategies for code smell removal within this specific ecosystem.

9 | CONCLUSION

This paper proposes a mapping between the code smells [20] and the refactorings [21] cataloged by us for Elixir, indicating which refactorings may be useful in code transformations carried out to remove each code smell. To establish these relationships between the two catalogs, we manually compared the characteristics of the problems caused by

each code smell with the motivations and the code improvements each of the Elixir refactorings can generate. The relationships established between these constructs, which together form practical guidelines for removing code smells in Elixir using refactorings tailored to the language, were validated through a survey of ten experienced developers drawn from diverse cultural backgrounds, genders, and companies.

We summarize the contributions of this paper as follows:

- We found that all 35 code smells for Elixir are covered by at least one of the refactorings in our catalog, meaning there is at least one refactoring that helps in the removal of these smells.
- We showed that some refactoring operations cataloged for Elixir can be useful for addressing more than one code smell, thereby highlighting their versatility in solving these issues.
- On the other hand, we found that 12 of the 82 refactorings cataloged for Elixir are not associated with the removal of known code smells for this language.
- We identified five new composite refactorings that can be useful in removing code smells in Elixir. Three of them can be used in conjunction with other refactorings to assist the removal of the smells they were mapped to.
- We showed that, according to experienced developers, the vast majority of code smells in Elixir (82.9%) can be largely or completely removed through sequences of refactorings tailored to the language.
- We have found evidence suggesting the existence of an uncatalogued *Design-related* smell for Elixir.
- Finally, we found that traditional refactorings proposed by Fowler and Beck [6] to improve the quality of object-oriented systems are also highly important for removing code smells in Elixir.

These findings have practical implications. For example, the mapping between catalogs conducted in this study can guide developers, especially those beginners to Elixir, on how to systematically remove code smells and improve the internal quality of systems implemented in this language. Additionally, this guide can serve as inspiration for removing code smells in systems implemented in other functional languages.

Finally, our research highlights the need to validate with Elixir developers the prevalence and relevance of the newly identified code smell (*i.e.*, *Unnecessary Explicit Recursion*), which emerged from the analysis of unmapped refactorings. It also points to the need for further research into the existence of other *Elixir-specific* composite refactorings. Although we identified five in this study, they were not the primary focus of our investigation.

Moreover, as indicated by our survey, future investigations are needed to identify more effective approaches for removing the *Inappropriate Intimacy* smell in Elixir, since the strategy proposed in this work still appears to be insufficient.

acknowledgements

José Valim deserves particular thanks for facilitating our communication with the Elixir community and supporting the Research with Elixir initiative (`http://pesquisecomelixir.com.br/`, in Portuguese). This research is supported by a grant from Dashbit: `https://dashbit.co/` and Rebase: `https://rebase.com.br/` and also by a grant from FAPEMIG.

data availability statement

The data that support the findings of this study are openly available in Zenodo at `https://doi.org/10.5281/zenodo.13835771`.

conflict of interest

The authors declare that they have no conflicts of interest.

references

[1] Thomas D. Programming Elixir |> 1.6: functional |> concurrent |> pragmatic |> fun. 1 ed. Pragmatic Bookshelf; 2018.

[2] Jurić S. Elixir in action. 3 ed. Manning; 2024.

[3] Almeida U. Learn functional programming with Elixir: new foundations for a new world. 1 ed. Pragmatic Bookshelf; 2018.

[4] Yamashita A, Moonen L. To what extent can maintenance problems be predicted by code smell detection? an empirical study. Information and Software Technology 2013;55(12):2223–2242.

[5] Soh Z, Yamashita A, Khomh F, Guéhéneuc YG. Do code smells impact the effort of different maintenance programming activities? In: 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER); 2016. p. 393–402.

[6] Fowler M, Beck K. Refactoring: improving the design of existing code. 1 ed. Addison-Wesley; 1999.

[7] Fontana FA, Ferme V, Marino A, Walter B, Martenka P. 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); 2013. p. 260–269.

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

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

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

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

[12] Ferreira F, Valente MT. Detecting Code Smells in React-based Web Apps. Information and Software Technology 2023;155:1–16.

[13] Nardone V, Muse BA, Abidi M, Khomh F, Penta MD. Video game bad smells: What they are and how developers perceive them. ACM Transactions on Software Engineering and Methodology 2023;32(4):1–35.

[14] Ferreira F, Borges H, Valente MT. Refactoring React-based Web apps. Journal of Systems and Software 2024;215:1–36.

[15] Arnaoudova V, Constantinides C. Adaptation of refactoring strategies to multiple axes of modularity: Characteristics and criteria. In: 6th International Conference on Software Engineering Research, Management and Applications (SERA); 2008. p. 105–114.

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

[17] Nguyen HA, Nguyen HV, Nguyen TT, Nguyen TN. Output-oriented refactoring in PHP-based dynamic web applications. In: 29th IEEE International Conference on Software Maintenance (ICSM); 2013. p. 150–159.

[18] Sobrinho EVdP, De Lucia A, Maia MdA. A systematic literature review on bad smells–5 w's: which, when, what, who, where. IEEE Transactions on Software Engineering 2021;47(1):17–66.

[19] Abid C, Alizadeh V, Kessentini M, Ferreira TN, Dig D. 30 years of software refactoring research: a systematic literature review. ArXiv 2020;abs/2007.02194:1–23.

[20] Vegi LFM, Valente MT. Understanding code smells in Elixir functional language. Empirical Software Engineering 2023;28(102):1–32.

[21] Vegi LFM, Valente MT. Understanding refactorings in Elixir functional language. Empirical Software Engineering 2025;30(108):1–58.

[22] Li H, Thompson S, Orosz G, Tóth M. Refactoring with Wrangler, updated: Data and process refactorings, and integration with Eclipse. In: 7th ACM SIGPLAN Workshop on ERLANG; 2008. p. 61–72.

[23] Li H, Thompson S. Refactoring support for modularity maintenance in Erlang. In: 10th IEEE Working Conference on Source Code Analysis and Manipulation (SCAM); 2010. p. 157–166.

[24] Li H, Thompson S. Clone detection and removal for Erlang/OTP within a refactoring environment. In: ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM); 2009. p. 169–178.

[25] Seijas PL, Thompson S. Identifying and introducing interfaces and callbacks using Wrangler. In: 28th Symposium on the Implementation and Application of Functional Programming Languages (IFL); 2016. p. 1–13.

[26] Thompson S, Li H, Schumacher A. The pragmatics of clone detection and elimination. The Art, Science, and Engineering of Programming 2017;1(2):1–34.

[27] Brown CM, Thompson S. Clone detection and elimination for Haskell. In: ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM); 2010. p. 111–120.

[28] Aranda M, Oliveira N, Soares E, Ribeiro M, Romão D, Patriota U, et al. A catalog of transformations to remove smells from natural language tests. In: 28th International Conference on Evaluation and Assessment in Software Engineering (EASE); 2024. p. 7–16.

[29] Brito A, Hora A, Valente MT. Towards a catalog of composite refactorings. Journal of Software: Evolution and Process 2023;1:1–22.

[30] Sousa L, Cedrim D, Garcia A, Oizumi W, Bibiano AC, Oliveira D, et al. Characterizing and identifying composite refactorings: Concepts, heuristics and patterns. In: 17th International Conference on Mining Software Repositories (MSR); 2020. p. 186–197.

[31] Scott ML. Programming Language Pragmatics. 4 ed. Morgan Kaufmann; 2015.

[32] Li H, Thompson S. A domain-specific language for scripting refactorings in Erlang. In: de Lara J, Zisman A, editors. Fundamental Approaches to Software Engineering (FASE), vol. 7212; 2012. p. 501–515.

[33] Shetty G, Sharma T. Mapping code smells and refactorings accurately: Insights from an empirical study. In: 19th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM); 2025. p. 1–11.

[34] Tsantalis N, Ketkar A, Dig D. RefactoringMiner 2.0. IEEE Transactions on Software Engineering 2022;48(3):930–950.

[35] Sharma T. Multi-faceted code smell detection at scale using DesigniteJava 2.0. In: 21st IEEE/ACM International Conference on Mining Software Repositories (MSR); 2024. p. 284–288.

[36] Fördős V, Tóth M. Identifying code clones with RefactorErl. Acta Cybernetica 2016;22(3):553–571.

[37] Sharma T, Suryanarayana G, Samarthyam G. Challenges to and solutions for refactoring adoption: An industrial perspective. IEEE Software 2015;32(6):44–51.

[38] Bibiano AC, Soares V, Coutinho D, Fernandes E, Correia JaL, Santos K, et al. How does incomplete composite refactoring affect internal quality attributes? In: 28th International Conference on Program Comprehension (ICPC); 2020. p. 149–159.

[39] Cedrim D, Garcia A, Mongiovi M, Gheyi R, Sousa L, de Mello R, et al. Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects. In: 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE); 2017. p. 465–475.

[40] Pereira G, Vegi LFM, Di-Iorio V. Building refactoring tools: Insights from RefactorEx. In: 1st Workshop on Software Engineering for Functional Programming (SE4FP); 2025. p. 9–14.

[41] Vegi LFM, Valente MT. Towards a catalog of refactorings for Elixir. In: 39th International Conference on Software Maintenance and Evolution (ICSME); 2023. p. 358–362.

[42] Humble J, Farley D. Continuous delivery: Reliable software releases through build, test, and deployment automation. 1 ed. Addison-Wesley Professional; 2010.

[43] Gamma E, Helm R, Johnson R, Vlissides J. Design patterns: Elements of reusable object-oriented software. 1 ed. Pearson Education; 1994.

[44] Gwet KL. Handbook of inter-rater reliability. 4 ed. Advanced Analytics; 2014.

[45] David HA. In: Gini's Mean Difference John Wiley & Sons, Ltd; 2006. .

[46] Krippendorff K. Content analysis: An introduction to its methodology. 4 ed. SAGE Publications; 2018.

author biographies

Lucas Vegi completed his Ph.D. in Computer Science at the Federal University of Minas Gerais (UFMG) in late 2024. Since 2015, he has served as an Assistant Professor at the Federal University of Viçosa (UFV), where he also earned his M.Sc. in Computer Science. His research interests include functional programming, software quality, reuse, maintenance, and evolution. Contact him at lucas.vegi@ufv.br or www.dpi.ufv.br/prof-lucas-francisco-da-matta-vegi/.

Marco Túlio Valente is an Associate Professor in the Computer Science Department at the Federal University of Minas Gerais (UFMG), where he also heads the Applied Software Engineering Research Group (ASERG). His research interests include software quality, maintenance, and evolution. He is a Researcher I-D of the Brazilian National Research Council (CNPq) and holds a Researcher from Minas Gerais State scholarship from FAPEMIG. Contact him at mtov@dcc.ufmg.br or www.dcc.ufmg.br/~mtov.

A | APPENDIX - REFACTORING CODE SMELLS: PRACTICAL GUIDELINES

In this appendix, we provide a detailed explanation of how the removal of each of the 35 code smells for Elixir can be assisted by refactoring strategies. Along with 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 they are part of a sequence of operations. Specifically, when a refactoring should be used alone, it is listed with a bullet point (*i.e.*, •) in this appendix. Conversely, when a refactoring forms part of a sequence of operations, it is listed using numbering to define its order (*e.g.*, 1.; 2.; etc.)

This appendix is organized with a section for each code smell, arranged in alphabetical order. This document can **guide developers, especially those beginners to Elixir, on how to systematically remove code smells** and improve the internal quality of their systems implemented with this language. More details about each code smell and refactoring, including code examples, are available at `https://github.com/lucasvegi/Elixir-Code-Smells` and `https://github.com/lucasvegi/Elixir-Refactorings`, respectively.

A.1 | Accessing non-existent map/struct fields

- *Default value for an absent key in a Map*: When trying to access the value of a key from a `Map` dynamically, it is not possible to determine if a key is non-existent or if it has an associated `nil` value. This refactoring can eliminate the smell, as the ambiguity in the returns of dynamic accesses will no longer occur after its application.
- *Introduce pattern matching over a parameter*: If the smell instance occurs in the function signature, such as in a guard clause, we can use this refactoring to extract the value of that field and then use it in the guard. This way, we can clearly determine if a field does not exist or if it simply has an associated `nil` value.
- *Simplifying checks by using truthness condition*: Optionally, when using dynamic access to fields of a `Map` and needing to return a default value for non-existent fields or those associated with `nil`, we can perform this refactoring to solve this issue, thus producing cleaner code.
- *Explicit a double boolean negation*: Optionally, if we are using double boolean negation to check if a dynamically accessed field in a `Map` exists, this refactoring can improve code readability by replacing the unintuitive logic with helper functions that utilize pattern matching.

1. *Struct field access elimination*: Optionally, if accesses to the same field, whether it exists or not, occur many times within a function, we can use this refactoring to replace these accesses with a temporary variable responsible for storing the value of that field.
2. *Equality guard to pattern matching*: Optionally, when a temporary variable extracted from a `struct` field is only used in an equality comparison in a guard, extracting and using that variable is unnecessary, as we can perform that equality comparison directly with pattern matching. To do this, we can use this refactoring.

A.2 | Agent obsession

1. *Generalise a function definition*: When functions responsible for directly interacting with the `Agent` are scattered throughout the system, it indicates the presence of this smell. We can refactor them simultaneously using this operation, centralizing the responsibility for interacting with the `Agent` in a single module or function.
2. *Moving a definition*: After generalizing the functions that were originally responsible for directly accessing the `Agent`, the generic function created to centralize this task might not be located in the most appropriate module. This refactoring can be used to address this issue.

3. *Add or remove a parameter*: Functions that were originally responsible for directly accessing the `Agent`, once generalized by the previous refactoring, may require the addition of new parameters to be passed to the generic function called within their bodies.

- *Behaviour extraction*: One way to remove this smell is to define a `behaviour` containing a contract that specifies the format of all functions intended to interact with an `Agent` throughout the system. Following this refactoring, all modules that wish to access the `Agent` must implement this extracted `behaviour`.

A.3 | Alternative return types

1. *Introduce a temporary duplicate definition*: When a function receives a `Keyword list` as a parameter, which can drastically change its return type depending on its contents, we should initially use this refactoring to create a copy of the original function for each different return type.
2. *Rename an identifier*: After creating the copies, we should rename each one according to its respective return type. Additionally, the bodies of the copies should be modified to fit the specific return types.
3. *Explicit a changed function signature* (Composite)
   - 3.1. *Add or remove a parameter*: At this point in the refactoring process, since the `Keyword list` parameter is no longer necessary in any of the renamed copies of the original function, we can use this operation to remove the unnecessary parameter.
   - 3.2. *Typing parameters and return values*: Finally, we can use this operation on each of the functions involved in this composite refactoring to document their interfaces.
4. *Remove dead code*: This refactoring can be used on the copies of the functions created previously in this sequence of transformations to clean up their bodies, removing unused code.

A.4 | Code organization by process

1. *Remove processes*: When code unnecessarily uses a process for organizational purposes where a simple module with functions would suffice, this refactoring can be used to remove the unnecessary concurrent processes and replace them with regular Elixir modules.
2. *Remove dead code*: When we transform a process into a regular Elixir module, some functions that previously implemented process callbacks (*e.g.*, `GenServer`) may become unnecessary and can therefore be removed using this refactoring.

A.5 | Comments

- *Extract function*: If a comment explains a block of code, that block can be extracted into a separate function. The name of the new function can often be derived from the comment itself.
- *Extract expressions*: If a comment is intended to explain a complex expression, the expression should be split into understandable sub-expressions using this refactoring.
- *Extract constant*: If a comment is used to explain magic numbers, this refactoring can replace the comments with constants that have human-friendly names.
- *Rename an identifier*: If a function, expression, or constant has already been extracted, but comments are still necessary to explain what they do, give a self-explanatory name to them using this operation.
- *Typing parameters and return values*: If a comment is used to document the types of a function's parameters or

the type of its return value, that comment can be replaced by a function specification using the `@spec` module attribute.

- *Add type declarations and contracts*: Alternatively, if a comment is used to document data structures received as parameters of a function or even returned by a function, this comment can be replaced by a type specification using the `@type` and `@typedoc` module attributes.

A.6 | Compile-time global configuration

- *Extract constant*: To remove this smell caused by using *Application Environment* in compile-time to define module's attributes (*i.e.*, constants), we can perform this refactoring in reverse, that is, perform an *inline* operation.
- *Introduce a temporary duplicate definition*: We can also use this refactoring to create a copy of the compile-time defined constant and replace its definition with a call to `Application.compile_env/3` instead of `Application.fetch_env!/2`.

1. *Folding against a function definition*: Another possibility would be to replace the location where a compile-time defined constant is used with a call to the *Application Environment* function responsible for defining the constant's content. This can be done using this refactoring.
2. *Remove dead code*: Finally, if the compile-time defined constant is no longer used in the module, we can simply remove it.

A.7 | Complex branching

- *Extract function*: When a function uses a conditional statement with many different branches, each responsible for handling a specific error type, we can use this refactoring to delegate each branch (*i.e.*, handling of a response type) to a different new private function. This approach makes the code cleaner, more concise, and readable.
- *Introduce pattern matching over a parameter*: Another possibility is to use this refactoring to break down complex branching into a multi-clause function, where each clause handles a different error type. This approach enhances readability and maintainability by organizing the code according to distinct error scenarios.

A.8 | Complex else clauses in with

1. *Extract function*: When an `else` clause in a `with` statement is used to handle different types of errors that may occur during the execution of the `with` clauses, the code can become confusing. To address this issue, we can use this refactoring to transform expressions in the `with` clauses into separate private functions. Each of these private functions will handle a specific error type, decentralizing the error-handling task.
2. *Remove dead code*: After extracting the new functions responsible for decentralizing error handling, the `with` statement will no longer need an `else` clause. Therefore, we can use this refactoring to remove the `else` clause.

- *Remove redundant last clause in "with"*: Optionally, if the last clause of the `with` statement used to chain operations is redundant, we can use this refactoring to make the code less verbose and more readable.
- *Moving "with" clauses without pattern matching*: Optionally, if the `with` statement does not perform pattern matching in the first and/or last clauses, we can use this refactoring to make the code more idiomatic and readable.

A.9 | Complex extractions in clauses

- *Simplifying pattern matching with nested structs*: When using pattern matching to perform deep extraction in nested `structs` passed as parameters to a clause of a multi-clause function, we may create unnecessarily messy and hard-to-understand code. With this refactoring, we can simplify this kind of extraction by performing pattern matching only on the outermost `struct` in the nesting, instead of matching patterns with very internal `structs`.
- *Converts guards to conditionals*: To prevent complex extractions in clauses, which are used both to access data extracted in guard clauses and in the function body, from making the code confusing, we can use this refactoring to replace all guards with traditional conditionals, consolidating them into a single clause for the function. This way, all extracted data will be used exclusively in the function body.
- *Equality guard to pattern matching*: Optionally, if an unnecessary temporary variable is extracted from a `struct`'s field in a clause of a multi-clause function and is only used for an equality comparison in a guard, this refactoring can simplify the pattern matching in that clause.
- *Struct guard to matching*: Optionally, if some clauses of a multi-clause function use guard clauses involving the functions `is_struct/1` or `is_struct/2`, this refactoring can simplify the pattern matching performed by these clauses.
- *Remove unnecessary calls to length/1*: If a list extraction is performed in a function clause only to use it in an unnecessary call to `length/1` in a guard clause, this extraction can be replaced by using pattern matching directly, eliminating the need for the guard clause. To do this, use this refactoring.
- *Function clauses to/from case clauses*: When complex extractions are done in clauses of multi-clause functions, making it difficult to understand which data is used inside or outside the function body, we can use this refactoring to transform a multi-clause function into a single clause function. By doing this, we will map the function's clauses into clauses of a `case` statement, ensuring that all extractions occur within the function body.

1. *Introduce a temporary duplicate definition*: When we are moving an extraction performed in a function clause into its body, we can initially use this refactoring to duplicate within the function body an extraction performed in the signature.
2. *Temporary variable elimination*: After duplicating the complex extraction within the function body, we can use this operation to remove unnecessary extracted parts, thereby cleaning up the code.

A.10 | Data manipulation by Migration

1. *Module decomposition* (Composite)
   - **1.1.** *Splitting a large module*: When a module that behaves like `Ecto.Migration` performs both data and structural changes in a database schema, it becomes less cohesive, more difficult to test, and therefore more prone to bugs. In these cases, we should split this module into two, moving only the attributes and functions related to data updates to the new module.
   - **1.2.** *Rename an identifier*: In some cases, after splitting an original module into several smaller and more cohesive modules, the name of the original module can no longer make sense, providing an opportunity also to perform this refactoring.
2. *Extract to outside* (Composite)
   - **2.1.** *Extract function*: We can use this operation in the original module to create a new function responsible for calling the routines for altering the data in the database, now present in the new module.
   - **2.2.** *Moving a definition*: After extracting this function, we can use this refactoring to reposition it in the new

module created after splitting the original module.

3. *Remove dead code*: Finally, we can eliminate the call to the extracted function in the original module to start the database alteration routines, as this function, now present in the new module, should only be called during the initialization of a `Mix.Task`.

- *Simplifying Ecto schema fields validation*: Optionally, we can take the opportunity to apply this refactoring in the original module, making it less prone to errors during the validation of the database schema modified via `Ecto.Migration`.
- *Pipeline for database transactions*: Optionally, we can also take the opportunity to apply this operation in the new module created only to perform changes on data, thus improving its readability by using `Ecto.Multi`.

A.11 | Divergent change

- *Module decomposition* (Composite)
  1. *Splitting a large module*: This smell can be removed by creating new cohesive modules and moving related functions into them.
  2. *Rename an identifier*: In some cases, after splitting an original module into several smaller and more cohesive modules, the name of the original module can no longer make sense, providing an opportunity to also apply this refactoring.
- *Moving a definition*: If there is already a module `B` that is more suitable to accommodate a function that makes module `A` less cohesive, we can simply move this function from module `A` to module `B` to remove this smell.
- *Behaviour extraction*: This smell can be removed by creating a new cohesive module `B` and moving related functions of module `A` into it. Thereby, we can use this refactoring to transform module `A` into a *behaviour definition* and create module `B` as a *behaviour implementation* of module `A`, thus achieving this goal.

A.12 | Duplicated code

- *Defining a subset of a Map*: It can be used to eliminate duplicated code generated by the manual access of values when extracting part of a `Map`.
- *Modifying keys in a Map*: This refactoring can be used to eliminate duplicated code generated by the manual process of replacing a name given to a `Map` key.
- *Folding against a function definition*: When code contains expressions that perform operations already implemented in existing functions, we can remove this code duplication by using this refactoring, thus replacing the own expressions with calls to the existing functions.
- *Extract expressions*: When the same expression is repeated multiple times within a function, we can assign the expression to a variable and reuse that variable in all parts where the result of the expression is needed.
- *Extract function*: When the same code block appears in two different functions, we can extract it to a new function and call this function from both places where the code was originally duplicated.
- *Reducing a boolean equality expression*: When dealing with a boolean expression consisting of multiple equality comparisons involving the same variable and logical `OR` operators, we can eliminate duplicated code using this refactoring, thus utilizing the `IN` operator and a list containing all possible valid values for the variable.
- *Generalise a function definition*: When we have different functions that have non-identical but equivalent expressions, we can use this refactoring to create a new higher-order function that generalizes the equivalent expressions and subsequently is called in places where the expressions were originally used.

- *Turning anonymous into local functions*: When we encounter the same anonymous function being defined in different points of the codebase, these anonymous functions should be transformed into a local function, and the locations where the anonymous functions were originally implemented should be updated to use the new local function.
- *Merging multiple definitions*: There are situations where a codebase may have distinct and complementary functions. Because they are complementary, these functions may have identical code snippets. When identified, these functions can be merged into a new function that will simultaneously perform the processing done by the original functions separately.
- *Move expression out of case*: When the same expression is repeated at the end of all branches of a `case` statement, this refactoring can be used to eliminate the duplicated code.
- *Remove redundant last clause in "with"*: When the last clause of a `with` statement is composed of a pattern identical to the predefined value to be returned by the `with` in case all checked patterns match, this clause is considered redundant. Therefore this duplicated code can be eliminated using this refactoring.
- *Static structure reuse*: When identical `tuples` or `lists` are used at different points within a function, they are unnecessarily recreated by Elixir. Use this refactoring to eliminate these redundant recreations by assigning the structures to variables, allowing them to be shared throughout the code.
- *Introduce import*: When a module `A` calls many functions from a module `B`, the name of module `B` may appear repetitively in the code of module `A` due to fully-qualified name calls. This type of duplicated code can be eliminated by importing module `B` into module `A`.
- *Widen or narrow definition scope*: When we encounter the same anonymous function defined in different parts of the codebase, these functions should be transformed into a local function, eliminating code duplication by expanding the scope of the original function. This refactoring serves as an alternative to *Turning anonymous into local functions*.

1. *Introduce Enum.map/2*: When each element of a `list` is manually generated by repeatedly calling the same function, we can use this refactoring to eliminate duplicated code and make the code more idiomatic.
2. *Transform to list comprehension*: Optionally, after using the previous refactoring to eliminate duplicated code, we can use this operation to convert the call to `Enum.map/2` into semantically equivalent code that can be also more declarative and easier to read.
3. *List comprehension simplifications*: Optionally, after using the previous refactoring as part of a sequence of atomic refactorings to eliminate duplicated code, we can also use this refactoring to revert the previous transformation, thereby retaining calls to the higher-order function `Enum.map/2` instead of using list comprehensions.

A.13 | Dynamic atom creation

1. *Extract function*: We can replace a call to the function `String.to_atom/1` with an explicit conversion. To do this, we can use this refactoring on the calls to `String.to_atom/1`, creating a new function. This new function should take a `string` as a parameter and convert it to an `atom`. The body of this function should include a conditional to check the content of the `string`. Depending on its content, the function will return a different `atom` directly.
2. *Introduce pattern matching over a parameter*: After extracting a new function for explicit conversions from `strings` to `atoms`, we can use this refactoring to transform the function into a multi-clause function, where each clause is responsible for returning one of the possible converted `atoms`.

- *Folding against a function definition*: If there is already a function in the module responsible for performing the explicit conversion of a `string` to an `atom` (*e.g.*, a function extracted for this purpose at a different point in the

same module), we can replace a call to `String.to_atom/1` with a call to that function.

- *Gradual change* (Composite)
  1. *Introduce a temporary duplicate definition*: Another alternative to refactor this code is to first duplicate the line where the function `String.to_atom/1` is called to create an `atom` dynamically. The new line should replace the call to `String.to_atom/1` with a call to `String.to_existing_atom/1`. This will ensure that string-to-atom conversions only map the `strings` to `atoms` already in memory. To enable this type of mapping, a suggestion is to create a `list` of these possible `atoms` within the function where this refactoring was applied.
  2. *Remove dead code*: After duplicating and modifying the duplicated line, we can eliminate the original line where the call to `String.to_atom/1` occurred.

A.14 | Feature envy

- *Extract to outside* (Composite)
  1. *Extract function*: If part of a function calls more functions from other modules than from the module where it is defined, we can use this refactoring to separate the envious part into a new function.
  2. *Moving a definition*: If a function calls more functions from other modules than from the module where it is defined (*e.g.*, the function extracted in the previous refactoring), we can move it to the module most accessed by it.
- *Remove import attributes*: By using this refactoring, we can directly identify the origin of a function being called by another function. This way, we can more clearly identify a *Feature envy* instance, helping us to subsequently remove it.

A.15 | GenServer envy

1. *Generalise a process abstraction*: When an `Agent` or `Task` goes beyond its suggested use cases and becomes painful, it is better to refactor it into a `GenServer` using this operation.
2. *Introduce processes*: When we finish generalizing a process, it may still be insufficient to achieve an optimal mapping with the parallel activities of the problem being solved. In these circumstances, we can use this refactoring to remove bottlenecks.
3. *Register a process*: When we create a new process, we can also use this refactoring to assign a user-defined name to the new process ID and use that user-defined name instead of the process ID in message passing.
4. *Remove dead code*: When we are generalizing a process `Agent` or `Task` into a `GenServer`, naturally, some functions of the module that represented the original process may become useless and can therefore be removed.

A.16 | Inappropriate intimacy

1. *Closure conversion*: A specific type of *Inappropriate intimacy* can be seen in closures, which are impure anonymous functions, as they access variables outside their scope. One way to remove this smell is by transforming the closure into a pure anonymous function.
2. *Add or remove a parameter*: Imagine the scenario where an anonymous function `A` is defined within a named function `B`. Furthermore, consider that `A` accesses a variable in its body that is a parameter of `B`, which is not passed as a parameter to `A` (*i.e.*, function `A` is a closure). When applying the previous refactoring to remove the *Inappropriate intimacy* instance in `A`, we may end up with unused parameters in the named function `B`. Therefore,

we can use the present refactoring in `B` to fix it.

- *Moving a definition*: If a function in module `A` is impure because it accesses internal details of module `B` without receiving them through its parameters, we can remove this smell by moving the function from `A` to `B`, thus reducing the coupling between modules.
- *Module decomposition* (Composite)
  1. *Splitting a large module*: If the modules involved in an instance of this smell have common interests, we can use this refactoring to put their commonality in a new module and make them high-cohesive and low-coupling modules.
  2. *Rename an identifier*: In some cases, after splitting an original module into several smaller and more cohesive modules, the name of the original module can no longer make sense, providing an opportunity to also apply this refactoring.

A.17 | Large class

- *Module decomposition* (Composite)
  1. *Splitting a large module*: When a module does the work of two or more, it becomes large, poorly cohesive, and difficult to maintain. In these cases, we should split this module into several new ones, moving to each new module only the attributes and functions with purposes related to their respective goals.
  2. *Rename an identifier*: In some cases, after splitting an original module into several smaller and more cohesive modules, the name of the original module can no longer make sense, providing an opportunity to also apply this refactoring.
- *Behaviour extraction*: This operation is helpful if it's necessary to have a list of operations and behaviors that client modules can reuse, thus reducing their sizes.
- *Moving a definition*: When a function is defined in module `A` but is more suited to the responsibilities of module `B`, we can move it to `B` to reduce the size of `A`.

A.18 | Large code generation by macros

- *Extract to outside* (Composite)
  1. *Extract function*: When we have a `macro` that generates a large volume of code, potentially compromising compiler performance, we can use this refactoring to extract part of the `macro`'s code and encapsulate it in a conventional function that the `macro` will call. This approach reduces the amount of code that is expanded and compiled with each invocation of the `macro`.
  2. *Moving a definition*: After extracting the function, it may eventually be necessary to move it to another module to make the code more cohesive.

A.19 | Large messages

- *Defining a subset of a Map*: If we are originally sending a complete `Map` from one process to another, but actually only need to send a few fields from this `Map`, we can use this refactoring to help reduce the size of the message sent.
- *Extract expressions*: When we use `spawn/1` to perform message passing between processes, we can pass an anonymous function as a parameter to `spawn/1` that accesses a `Map` field in its body. This will still copy over all of the

`Map`, because the `Map` variable is being captured inside the spawned function. The function then extracts the field, but only after the whole `Map` has been copied over. Suppose we only need to send one field of a `Map` between processes. In that case, we can reduce the size of this message by using this refactoring to store only the necessary value of the `Map` field in a temporary variable. This variable is then used in the spawned function.

- *Add a tag to messages*: Optionally, when we are removing this code smell, we may have the opportunity to adapt the processes that communicate with each other by adding tags that identify groups of messages exchanged between them.

A.20 | Long function

- *Extract function*: If a comment is needed to explain some part of the function body, this refactoring can be used to create a new function to be called at the location of this comment, thus decreasing the size of the original function.
- *Transform nested "if" statements into a "cond"*: If a function uses many nested `if` conditionals, this can greatly increase its size. In these cases, we can use this refactoring to decrease the size of the function.
- *Folding against a function definition*: If a function performs operations that can be delegated to other existing functions, it may become unnecessarily long. In these cases, we can use this refactoring to reduce the size of the original function.
- *Remove dead code*: If a function contains code that is no longer used, it may become unnecessarily long. In these cases we can use this refactoring to reduce the size of the original function.
- *Simplifying checks by using truthness condition*: When we know that a given data item can be `nil` and we need to return a default value if it is indeed `nil`, we can use this refactoring to reduce the number of lines in a function while maintaining clean and self-explanatory code.
- *Generalise a function definition*: When different functions have equivalent expression structures, these equivalent expressions can be generalized into a new higher-order function, thus reducing the size of the original functions.
- *Introduce pattern matching over a parameter*: When a function has many branches in its body that depend on values received as parameters, it can become unnecessarily long. We can use this refactoring to create short multi-clauses for the original function, where each clause handles a different branch.
- *Replace pipeline with a function*: When a function is unnecessarily long due to a pipeline of function calls that can be replaced by a single call, we can use this refactoring to address the issue.
- *Default value for an absent key in a Map*: A function can have its number of lines reduced by using this refactoring to replace the use of `if` statements that check the return of `Map.has_key?/2` with a call to the `Map.get/3` function.
- *Defining a subset of a Map*: When a function is unnecessarily long because it manually creates a subset of a `Map` by individually accessing each of the desired key/value pairs, we can use this refactoring to delegate this task to a call to `Map.take/2`.
- *Pipeline using "with"*: When a function uses nested conditionals solely to control a sequence of function calls, it can become long. By using this refactoring, we can make the function more idiomatic and reduce its number of lines of code.
- *Remove redundant last clause in "with"*: This refactoring can reduce the number of lines of code used by a `with` statement and consequently shorten the size of the function that uses this statement.
- *Modifying keys in a Map*: When a function is unnecessarily long because it manually replaces a `Map` key with a new key using `Map.get/2`, `Map.put/2`, and `Map.delete/2` functions together, we can use this refactoring to delegate this task to a call to `Map.new/2`, thus significantly reducing the volume of lines of code.

- *Remove nested conditional statements in function calls*: When a function uses unnecessary nested conditional statements, it can become long and less readable. In such circumstances, we can use this refactoring to replace the unnecessary nested conditional statements with less bulky code that maintains the same behavior, thus making the code simpler and more readable.
- *Splitting a definition*: When the refactoring *Merging multiple definitions* is used carelessly, there is a risk of creating a long function. In such cases, we can undo this operation using the present refactoring, thereby generating smaller, separate functions.
- *Merging match expressions into a list pattern*: If a function uses many lines of code with expressions that assign results to temporary variables but can be replaced by a single `list` generated through pattern matching, we can use this refactoring to reduce the size of the function.

1. *Convert nested conditionals to pipeline*: When a function uses nested conditionals solely to control a sequence of function calls, it can become long. By using this refactoring, we can give the function a more functional appearance and reduce the number of lines of code.
2. *Replace function call with raw value in a pipeline start*: When using the previous refactoring to remove this smell, we may also encounter an opportunity to apply the present operation, making the refactored code more idiomatic.

A.21 | Long parameter list

- *Add or remove a parameter*: If a function parameter for some reason is never used, it can be removed using this refactoring.
- *Reorder parameter*: When a function has a long list of parameters that reduces its readability, we can at least reorder the list into a more logical sequence to improve clarity.
- *Introduce parameter struct* (Composite)
  1. *Grouping parameters in tuple*: If a function has sequential and related parameters in its list, these parameters can be grouped into a `tuple`, thus reducing the length of the list.
  2. *From tuple to struct*: A `tuple` used to group parameters can eventually be replaced by a `struct` using this refactoring.

A.22 | Modules with identical names

- *Rename an identifier*: In Elixir, there is a naming convention for modules that should be followed when implementing libraries. According to this convention, a library should use its own name as a prefix (namespace) for all its module names (*e.g.*, `LibraryName.ModuleName`). When a library does not adhere to this naming convention, it can lead to name conflicts for the library's clients. To remove this smell, we can rename the modules of a library to adapt them to this convention. It is important to be careful when performing this refactoring on already released libraries, as it may cause breaking changes in client code.
- *Gradual change* (Composite)
  1. *Introduce a temporary duplicate definition*: As explained in the previous refactoring, when a library does not follow the naming convention for modules, it can lead to name conflicts in their clients. To remove this smell in already released libraries, we can use this present refactoring on the modules of a library, adapting the names of the copies to this convention. Meanwhile, the modules with names outside the convention should be marked as deprecated but not immediately removed, to avoid breaking client code.
  2. *Remove dead code*: When modules deprecated by the previous refactoring have remained deprecated long

enough for clients to adapt to the new naming conventions, we can use this refactoring to permanently eliminate them, thereby removing the risk of name conflicts.

- *Move file*: If the same directory contains two modules with the same name defined in different files, we can use this refactoring to relocate one of these modules to a new directory. Naturally, we will also need to rename the moved module to conform to the naming convention for modules in Elixir, which recommends using the directory name as a prefix (namespace) for all module names contained within it (*e.g.*, `LibraryName.ModuleName`).

A.23 | Primitive obsession

- *Introduce parameter struct* (Composite)
  1. *Grouping parameters in tuple*: If primitive/basic type values are used in function parameters to inadequately represent more complex real-world abstractions, you can initially apply this refactoring to group them.
  2. *From tuple to struct*: A `tuple` used to group parameters can eventually be replaced by a `struct`, thus creating a more robust data structure for this purpose.
- *Add type declarations and contracts*: We can use this refactoring to generate a type specification to replace primitive/basic values.

A.24 | Shotgun surgery

- *Moving a definition*: When we need to simultaneously make a series of small changes in different modules, it's easy to overlook something important. In this case, you can use this refactoring to reorganize the modules, aiming to make them more cohesive so that all necessary changes are concentrated within their respective modules. If no current module seems like a good candidate to receive parts moved from others, we need to create one.

A.25 | Speculative assumptions

- *Introduce pattern matching over a parameter*: This smell arises when developers write defensive or imprecise code, which can return incorrect values that were not planned for. To remove it, we can use this refactoring to force a function to crash instead of returning an invalid value when something unexpected happens.
- *Pipeline using "with"*: If this smell occurs within nested conditional statements, we can use this refactoring to remove them. This operation relies on pattern matching to prevent the continuation of tasks that depend on specific data formats.

A.26 | Speculative generality

- *Inline function*: Use this refactoring to get rid of unused functions or even unnecessarily created ones.
- *Inline macro*: Similarly to the previous refactoring, use this operation to eliminate unused macros or those that were unnecessarily created.
- *Add or remove a parameter*: Functions with unused parameters should be reviewed using this refactoring.
- *Rename an identifier*: Functions with abstract names should be renamed using more specific names that reflect their current task, rather than potential future tasks they might perform.
- *Behaviour inlining*: Unnecessary delegation/generalization caused by the excessive use of Elixir's `behaviour` can be removed with this refactoring.

- *Remove dead code*: In general, any code created unnecessarily to support future features that are never implemented can be refactored using this operation.
- *Eliminate single branch*: When a conditional is created to potentially provide different treatments for different types of data but actually provides the same treatment to all, we can simplify the code by removing unnecessary complexity.

A.27 | Switch statements

1. *Replace conditional with polymorphism via Protocols*: Suppose the same sequence of conditional statements appears duplicated in the code. In that case, we may be forced to make changes in multiple parts of the code whenever a new check needs to be added to these duplicated sequences of conditional statements. This refactoring introduces polymorphism to data structures, thus improving the code's extensibility to handle flow controls based on data types.
2. *Converts guards to conditionals*: The inverse operation of this present refactoring can be used to complement the previous refactoring operation, combining polymorphism with guard clauses in the implementation of the functions defined in the created `Protocol`.

- *Introduce pattern matching over a parameter*: If a switch/conditional of a function is based on a set of numbers or strings that form a list of allowable values for some parameter (*i.e.*, "type code"), we can use this refactoring to replace the conditional with a multi-clause function.
- *Introduce overloading*: We can overload a function, transforming it into a multi-clause function, to eliminate a conditional.

1. *Extract to outside* (Composite)
   1.1. *Extract function*: Often the duplicated switch/conditional statement switches on a "type code". To isolate a switch/conditional in a *type code host*, start by extracting one of the duplicated switch/conditional statements into a new function.
   1.2. *Moving a definition*: Second, the extracted function should be moved to the right module.
2. *Generalise a function definition*: After the previous composite refactoring, the moved function should be transformed into a higher-order function. One of the parameters of this generalized function will receive a function as an argument, responsible for defining the strategy of the internal conditional check.
3. *Folding against a function definition*: Finally, all points in the code with duplicated switch/conditional statements can use this refactoring to replace the duplicated code with calls to the previously defined higher-order function.

A.28 | Unnecessary macros

- *Inline macro*: When code is implemented as a `macro` but could be implemented as a conventional function in Elixir, we can use this refactoring to remove this smell, improving the readability of the code.
- *Extract to outside* (Composite)
  1. *Extract function*: When creating a `macro` is unavoidable, but part of it could be implemented as a conventional named function, we can extract this part of the macro's code and encapsulate it into a conventional function, which the `macro` will then call. This approach improves code organization and readability while leveraging the macro for its specific role.
  2. *Moving a definition*: After extracting the function, it may eventually be necessary to move it to another module to make the code more cohesive.

A.29 | Unrelated multi-clause function

1. *Rename an identifier*: A possible solution to this smell is to use this refactoring to break up the business rules that are mixed into several different simple functions. Specifically, groups of clauses from the original function that share related functionalities can be renamed with an identical name, thus forming a new multi-clause function.
2. *Function clauses to/from case clauses*: Optionally, after renaming the groups of related clauses, thus creating new multi-clause functions, these multi-clause functions can be transformed into single-clause functions by mapping function clauses into clauses of `case` statements.

- *Moving a definition*: When unrelated clauses of a multi-clause function differ to the point where they do not make sense in the current module, these clauses can be moved to other modules where they fit better, thus making the code more cohesive.
- *Struct guard to matching*: Optionally, when some of the clauses of a multi-clause function use a guard clause involving the functions `is_struct/1` or `is_struct/2`, we can use this refactoring to simplify the pattern matching performed by these function clause's signature.
- *Equality guard to pattern matching*: Optionally, when an unnecessary temporary variable is extracted from a `struct`'s field in a clause of a multi-clause function only to be used in an equality comparison in a guard, we can use this operation to simplify the pattern matching performed by this function clause's signature.
- *Simplifying guard sequences*: Optionally, when a clause of a multi-clause function contains redundant logical propositions, we can simplify the pattern matching performed by this function clause's signature through the transformation carried out by this refactoring.
- *Converts guards to conditionals*: Optionally, when what differentiates the clauses of a multi-clause function are only the logical checks performed in their guard clauses, we can use this refactoring to replace all guards with traditional conditionals, creating only one clause for the function.
- *Simplifying pattern matching with nested structs*: Optionally, when using pattern matching to perform deep extraction in nested `structs` passed as parameters to a clause of a multi-clause function, we may create unnecessarily messy and hard-to-understand code. With this refactoring, we can simplify this kind of extraction by performing pattern matching only on the outermost `struct` in the nesting, instead of matching patterns with very internal `structs`.
- *Remove unnecessary calls to length/1*: Optionally, when unnecessary calls to the function `length/1` are used in guard clauses to differentiate the pattern matching performed by different clauses of a multi-clause function, we can replace these calls with direct pattern matching on the parameter list of the function clauses, thereby improving code efficiency.

A.30 | Unsupervised process

- *Moving error-handling mechanisms to supervision trees*: Regardless of whether error-handling mechanisms are used or not, an unsupervised process can be moved to a supervision tree using this refactoring. With this transformation, the initialization of processes is delegated to a `Supervisor` and is no longer performed directly by the clients. This refactoring allows you to choose which module that behaves as a `Supervisor` to move to, or even to create a new module that implements the `Application` behaviour to act as a supervision tree.
- *Moving a definition*: We can move the process initialization function calls (*e.g.*, `GenServer.start/3`) into a `Supervisor`, delegating this task to it.

A.31 | Untested polymorphic behaviors

1. *Introduce overloading*: We can use this refactoring to transform a simple polymorphic function into a multi-clause function, where each clause of the function is responsible for handling one of the supported data types.
2. *Folding against a function definition*: After transforming a polymorphic function into a multi-clause function, we can use this refactoring to delegate part of the processing within a clause's body to calls of other clauses of the same function.

- *Typing parameters and return values*: Another improvement possibility for code suffering from this code smell is to use this refactoring to document a polymorphic function, making it clear which data types it supports.

A.32 | "Use" instead of "import"

1. *Introduce import*: When a `use` directive is unnecessarily used to establish a dependency between two modules, it can lead to unwanted propagation of internal dependencies from module `A` to module `B`. To remove this smell, we can initially replace a `use` directive with an `import` or `alias` directive. For this, we can first use this refactoring, which will create a more superficial dependency in module `B` with module `A`.
2. *Remove dead code*: After adding the `import` directive, we should remove the `use` directive using this refactoring.

- *Alias expansion*: Optionally, if the directive used to replace `use` is an `alias` and it is in the multi-alias format (*e.g.*, `alias Foo.Bar.{Baz, Boom}`), we can use this refactoring, thus providing an improvement in code readability and traceability.
- *Remove import attributes*: Optionally, if after replacing `use` with `import` we identify that a module is excessively importing other modules to the point of impairing readability—making it difficult to identify the origin of the functions it calls—we can perform this refactoring to some unnecessary imports. After that, we will then call the functions from the removed `imports` by their fully-qualified names.

A.33 | Using App Configuration for libraries

- *Explicit a changed function signature* (Composite)
  1. *Add or remove a parameter*: When we have a library function that depends on globally defined values, which reduces its reusability, we can use this refactoring to add a new optional parameter of type `Keyword list` with a default value. This new parameter allows the function to be configured in different ways when called. If the optional parameter is not provided in the call, the function will continue to behave as originally, using the same global configurations.
  2. *Typing parameters and return values*: We can also use this refactoring to explicitly document the type of the added parameter (*i.e.*, `Keyword list`).

A.34 | Using exceptions for control-flow

1. *Rename an identifier*: To prevent a library function from always forcing third-party code to handle an error as an exception, we can initially rename the original function, adding a trailing `!` at the end of its name. In Elixir, there is a convention where a `!` (*i.e.*, *trailing* or *bang*) at the end of a function name indicates that it may raise an exception.
2. *Introduce a temporary duplicate definition*: After renaming the original function by adding a `!` at the end of its name, we can use this refactoring to duplicate the renamed original function. This copy should have the original

function's name, without the `!` at the end. Instead of raising exceptions, it should return data in the `tuple` format (*i.e.*, `{:ok, _}` or `{:error, msg}`). The message provided in the error `tuple` should be the same as the one originally displayed in the exception version.

3. *Folding against a function definition*: Finally, we can use the present refactoring to change the *bang* variant (*i.e.*, the original function that raises an exception, with `!` at the end of its name). With this transformation, the raising version is implemented on top of the non-raising version of the code.

1. *Introduce processes*: Another possibility for removing this smell is, if the module where the function that raises an exception is not a process (*e.g.*, `GenServer` or `Task`), we can use this refactoring to transform the module into a process.
2. *Moving error-handling mechanisms to supervision trees*: When a process is using defensive programming (*i.e.*, `try..rescue`) to handle exceptions and even control the execution flow, we can use this refactoring to eliminate this type of handling, transitioning instead to the *"Let it crash"* style.

A.35 | Working with invalid data

1. *Typing parameters and return values*: When a library function does not validate the types of its parameters at its signature, we can at least use this refactoring to document these data. This will help the clients of this function (*i.e.*, third-party code) to protect themselves from potential errors caused by invalid data.
2. *Add type declarations and contracts*: If recurring data structures are found when documenting functions of a library, these structures can be named using the present refactoring, thus creating new reusable data types and increasing the system's readability.

- *Introduce pattern matching over a parameter*: Optionally, if a client validates the data passed to a library function call using traditional conditional statements, we can use this refactoring to make the client function more idiomatic while still addressing the smell.
- *Struct guard to matching*: Optionally, if a guard clause involving the functions `is_struct/1` or `is_struct/2` is used to avoid working with invalid data, we can use this refactoring to simplify the code used to remove this code smell.
- *Simplifying guard sequences*: Optionally, if a guard clause with redundancies is used to avoid working with invalid data, we can use the present refactoring to also simplify the code used to remove this code smell.
- *Converts guards to conditionals*: Optionally, we can replace guard clauses used in a multi-clause client function to handle invalid data with traditional conditional statements. By doing so, we can use this refactoring to consolidate all data type validations into a single-clause function.