Empirical Software Engineering (2025) 30:108
https://doi.org/10.1007/s10664-025-10652-y

Understanding refactorings in Elixir functional language

Lucas Francisco da Matta Vegi[1] · **Marco Tulio Valente**[1]

Accepted: 30 March 2025

Abstract

Elixir is a modern functional language known for its robustness and scalability. This language has seen a growing adoption by companies worldwide in the last 12 years. Despite this fact, and to the best of our knowledge, there are few works in the literature focused on studying refactoring strategies for code implemented with this language. In a preliminary and previous study, we conducted a systematic literature review to provide an initial list of refactorings for Elixir. Aiming to expand the results of this preliminary study, in this work we use a mixed methodology based on a broader systematic literature review, a grey literature review, and the mining of artifacts in GitHub repositories to prospect and document new refactorings for this language. As a result, we propose a comprehensive catalog of 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, which are also compatible with Elixir code. We validated this catalog by surveying 144 experienced Elixir developers from 42 countries spanning all continents. In this survey, we assessed the levels of relevance and prevalence of each refactoring in the catalog. We show that 93% of the refactorings in Elixir are at least moderately relevant, suggesting they can improve the code quality of Elixir systems. Furthermore, 71% of these refactorings are frequently used in production code. Our results have practical implications related to the learning and use of refactorings in Elixir.

Keywords Refactorings · Elixir · Functional programming · Systematic literature review · Grey literature review · Mining software repositories

1 Introduction

Historically, functional languages have not been as popular in the industry as object-oriented ones. However, there has been a recent increase in interest in functional languages in real-world applications (Bordignon and Silva 2020). More specifically, Elixir is a mod-

Communicated by: Ali Oun

✉ Lucas Francisco da Matta Vegi
lucasvegi@dcc.ufmg.br; lucasvegi@gmail.com

Marco Tulio Valente
mtov@dcc.ufmg.br; mtvalente@gmail.com

1 Department of Computer Science, UFMG, Belo Horizonte, Brazil

Published online: 07 May 2025

ern functional programming language that is gaining traction in the industry, with over 300 companies worldwide using the language, including Discord, Heroku, and PepsiCo.[1] This language is renowned for its performance in parallel and distributed computing environments (Thomas 2018). Conceived in 2012, Elixir draws inspiration from a blend of programming languages, such as Ruby, Haskell, Erlang, and Clojure (Jurić 2024). According to approximately 72% of developers who participated in the Elixir Survey 2023,[2] the three main factors that influenced their decisions to adopt Elixir are its functional paradigm, increased productivity, and facilitated support for concurrency. Furthermore, as shown by the Stack Overflow Survey,[3] Elixir is highly admired by developers who work with it. In the latest three editions of the survey (*i.e.,* 2022, 2023 and 2024), Elixir ranked second among the languages that developers currently use and intend to continue using in the next year.

Functional languages like Elixir have distinct characteristics compared to mainstream languages, which are typically imperative and object-oriented. Specifically, functional languages feature immutable data, support lazy evaluation, treat functions as first-class citizens, and often have pure functions that do not cause side-effects (Backfield 2014). Additionally, iterations in functional languages are commonly performed via recursive functions, as they typically lack classical iteration constructs like `while` and `do..while`, which are prevalent in object-oriented languages that rely on mutable state (Almeida 2015). Furthermore, functional languages utilize pattern matching, a mechanism that allows values to be compared with specific patterns while simultaneously extracting information from them (Scott 2015). Based on the identified patterns, different actions can be executed by the code. Consequently, this feature is frequently used as a control-flow mechanism instead of traditional conditional statements like `if..else` (Almeida 2015). When combined, this set of characteristics leads to unique design and code organization problems, as demonstrated in our previous work on code smells for Elixir (Vegi and Valente 2023).

Refactorings are code transformations that do not alter the code's behavior (Fowler and Beck 1999). According to Abid et al. (2020) and Silva et al. (2016), the pursuit of code quality improvement is one of the main motivations for developers to refactor their code. Therefore, just like in any programming language, it is natural to expect Elixir developers to also seek design enhancements in their codebase through refactoring strategies. However, although Elixir is becoming notably popular, there is a scarcity of studies conducted on refactoring for this language, as shown by Abid et al. (2020). Moreover, to the best of our knowledge, only one preliminary study conducted by us Vegi and Valente (2023b) has investigated refactorings specifically tailored for Elixir. Given the growing significance of functional languages, particularly Elixir, the existence of unique design and code organization problems in this language (Vegi and Valente 2023), and the lack of studies on how to refactor systems developed with it, this work aims to fill this gap by proposing a comprehensive catalog of refactorings for Elixir.

In a preliminary 5-page short paper (Vegi and Valente 2023b), we conducted a systematic literature review (SLR) to catalog refactoring strategies compatible with Elixir. We found 54 refactorings that have been proposed for other functional languages (*i.e.,* Clojure, Erlang, Haskell, and Lisp), as well as languages that inspired the creation of Elixir, such as Ruby. Therefore, this journal paper extends this preliminary study by using complementary methodologies to prospect, document, and validate refactorings for Elixir, thus creating a

[1] https://elixir-companies.com/

[2] https://curiosum.com/surveys/elixir-2023

[3] https://insights.stackoverflow.com/survey

comprehensive catalog. Specifically, we extend the previous short paper in the following ways:

1. In our preliminary short paper (Vegi and Valente 2023b), we focused on cataloging refactorings based on the analysis of 65 papers found in a simplified SLR. In our current work, we conducted a new SLR using an improved query string compared to the one used in the previous study. In addition to including a significant number of programming languages in this new SLR's query string, we also incorporated the term *"program transformation"*, since it is a synonym for refactoring commonly used in the context of functional languages (Burstall and Darlington 1977; Hudak 1989; Partsch and Steinbrüggen 1983). This enabled us to select 70 additional papers for analysis, leading to the identification of one novel refactoring compatible with Elixir.
2. We reviewed 26 grey literature documents concerning refactorings for Elixir. From these documents, we extracted refactoring strategies in 24 of them, including 21 novel ones for our catalog.
3. We conducted a mining study analyzing 119 artifacts (*i.e.,* issues, commits, pull requests, and code files) from the Top-10 most popular Elixir repositories on GitHub. Among these artifacts, 116 showed instances of code transformations, some of which resulted in six new refactorings for our catalog.
4. We have provided in a public GitHub repository (Vegi and Valente 2023a) an improved documentation of the 54 refactorings found in our previous study (Vegi and Valente 2023b). Additionally, we have included in this repository the documentation for the 28 new refactoring strategies cataloged in our current study. Among these, 14 are ELIXIR-SPECIFIC refactorings, a category that did not exist in our preliminary short paper.
5. We conducted a survey with 144 Elixir developers to validate the proposed catalog of refactorings. Each participant received a list of refactorings and they were asked to rank each one's relevance and prevalence on a scale of one (*very low*) to five (*very high*). To analyze our responses and quantify the prevalence and relevance levels of each refactoring, we used descriptive statistics. Additionally, we employed the non-parametric Mann-Whitney test (Siegel and Castellan 1988) to evaluate whether the respondents' experience level influenced their perception of the refactorings.

The contributions of this paper are threefold: (1) we cataloged in detail 82 refactorings for Elixir and categorized them into four different groups (ELIXIR- SPECIFIC REFACTORINGS, FUNCTIONAL REFACTORINGS, ERLANG- SPECIFIC REFACTORINGS, and TRADITIONAL REFACTORINGS); (2) we showed through the results of our survey that most of the cataloged refactorings (70.6%) are at least moderately prevalent, indicating they are common in production code. Furthermore, the vast majority of refactorings (92.7%) are at least moderately relevant, suggesting they have the potential to enhance the quality of systems developed in Elixir. Lastly, nine refactorings in the catalog (11%) have a high average score between their relevance and prevalence levels, indicating they deserve special attention from Elixir developers; (3) we found that the experience level of the developers who participated in our survey had little impact on their perceptions of the relevance and prevalence of the refactorings in our catalog, as only 19% of the refactorings were influenced by these factors. These findings have practical implications. For example, developers should prioritize mastering the most prevalent refactorings first, as understanding these code transformations can save time during the code review process. Additionally, developers should perform the most relevant refactorings first to maximize improvements in code quality.

The remainder of this paper is structured as follows. Section 2 provides an overview of the Elixir language and refactorings. Section 3 outlines our catalog of Elixir refactorings, along with potential threats to validity concerning the identification and documentation of refactorings. In Section 4, we delve into the details of the survey we conducted with Elixir developers to validate our catalog. This section also discusses potential validity issues, and describes the questionnaire design and the methods used to recruit respondents. Section 5 discusses the practical implications of our findings. Lastly, we present some relevant related work in Section 6, and in Section 7 we conclude the paper by presenting final remarks and suggestions for future work.

2 Background

2.1 Elixir Language

Elixir is a modern functional programming language renowned for its efficient performance in parallel and distributed computing environments. Although not yet mainstream, functional languages are becoming more popular in the industry, as seen by the growing number of users (Bordignon and Silva 2020). More than 300 companies worldwide—including some well-known ones like Adobe, Cabify, Discord, Motorola, and Pinterest—already utilize Elixir in their production code.

José Valim created this language in 2012, drawing inspiration from features found in various other languages like Erlang, Haskell, Clojure, and Ruby (Jurić 2024). Specifically, Elixir's syntax is based on Ruby, thus contributing to its friendly nature. It uses immutable data, similar to Haskell, which makes it well-suited for concurrent environments. Despite their differing syntaxes and features, Elixir code can seamlessly integrate with Erlang code since they both run on BEAM, which is Erlang's virtual machine. BEAM is renowned for its robustness, fault tolerance, and capability to handle concurrent and distributed systems (Almeida 2015). Moreover, Elixir inherits features like `protocols` and `macros` from Clojure, making it a polymorphic and extensible language. Elixir is primarily used to develop high-demand Web applications,[4] but it can also be used to develop machine learning systems,[5] embedded software,[6] code notebooks,[7] data science solutions,[8] and systems for many other purposes.

With Elixir, developers can create scalable and fault-tolerant concurrent systems more easily and with fewer computational resources. Instead of directly managing semaphores or other synchronization mechanisms, Elixir systems are based on units of concurrency known as BEAM processes (Thomas 2018). Unlike OS processes or threads, BEAM processes are lightweight concurrent entities managed by the VM. Creating a single BEAM process requires only a few microseconds, and its initial memory usage is minimal, typically just a few kilobytes. In contrast, OS processes typically consume a couple of megabytes (Jurić 2024).

Elixir programs are organized by `modules`, which are groups of functions. Listing 1 shows an Elixir module (`ListOperations`) composed of three functions—`empty?/1`, `sum_list/1` and `sort_double_list/1`. In Lines 22 to 24, these functions are called

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

[5] https://github.com/elixir-nx/nx

[6] https://www.nerves-project.org/

[7] https://livebook.dev/

[8] https://github.com/elixir-explorer/explorer

respectively using *Elixir's interactive shell* (IEx),[9] which is an intelligent terminal that allows developers not only to run their code but also to test and access its documentation.

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

```
1  defmodule ListOperations do
2    def empty?(list) when is_list(list) do
3      if length(list) > 0 do
4        false
5      else
6        true
7      end
8    end
9
10   def sum_list([]), do: 0
11   def sum_list([head | tail]) do
12     head + sum_list(tail)
13   end
14
15   def sort_double_list(list) do
16     list
17     |> Enum.map(fn x -> x * 2 end)
18     |> Enum.sort()
19   end
20 end
21 ...
22 iex(1)> ListOperations.empty?([2, 3, 1]) # false
23 iex(2)> ListOperations.sum_list([2, 3, 1]) # 6
24 iex(3)> ListOperations.sort_double_list([2, 3, 1]) # [2, 4, 6]
```

In Elixir, it is possible to use the `when` statement to define guard clauses, as seen in the definition of `empty?/1` (Line 2). A guard clause allows a function to perform conditional checks directly in its signature, thereby determining if the code within its body will be executed when the function is called. For example, this feature can be used to perform type validations on a function's parameters, which is very useful for a dynamically-typed language like Elixir.

Although Elixir has classical conditional constructs like `if..else` (Lines 3 to 7), throughout this paper, we will show that these constructs can be replaced by pattern matching, which, among other things, is often used as a control-flow mechanism in Elixir (Jurić 2024). The use of pattern matching in Elixir also allows the implementation of multi-clause functions, as shown in the definition of `sum_list/1` (Lines 10 and 11). This overloaded function has two clauses with the same name, both receiving a `list` as a parameter. When `sum_list/1` is called, the two clauses are matched against the parameter to determine which clause will be executed. Lists in Elixir have a recursive nature, where the first element is known as the *head* and the sub-list composed of the remaining elements is known as the *tail*. Therefore, when iterating through a list, we reach the end when its *tail* is an empty list, as shown in the pattern of the first clause of `sum_list/1` (Line 10).

Since Elixir does not have classical iteration constructs like `while` and `do..while`, the function `sum_list/1` (Lines 10 and 11) is defined as a recursive one, as recursion is the primary looping mechanism used in this language. However, Elixir has several high-level abstractions that hide the details of recursion, such as the functions `Enum.map/2` and `Enum.sort/1` (Lines 17 and 18). These functions are used to iterate over each element in the list received as a parameter by `sort_double_list/1` (Line 15). Note that `Enum.map/2`

[9] https://hexdocs.pm/iex/IEx.html

is a higher-order function because it receives an anonymous function (*i.e.,* `fn x -> x * 2 end`) as one of its parameters. This anonymous function is called by `Enum.map/2` for each element in the list received in its first parameter (Line 16). Therefore, `Enum.map/2` returns a modified list that is passed to the `Enum.sort/1` (Line 18).

The *pipe operator* (`|>`) is an idiomatic feature of Elixir. In our example, two pipe operators are used to chain the sequence of operations performed on the list received by `sort_double_list/1`. For example, in Elixir, `foo() |> bar(p)` is equivalent to `bar(foo(), p)`. For this reason, in the `Enum.map/2` call, only one parameter is directly provided instead of two (Line 17). For the same reason, in the `Enum.sort/1` call, no parameter is specified (Line 18). Finally, considering that data in Elixir is immutable, the lists returned by `Enum.map/2`, `Enum.sort/1`, and consequently by `sort_double_list/1` are modified copies of the original list received as a parameter by `sort_double_list/1`.

2.2 Refactorings

Refactoring is a widely recognized technique that enhances the design of a system by changing its code without changing its behavior, thus facilitating its evolution and maintenance (Fowler and Beck 1999). The term was first proposed by Opdyke (1992) and popularized by Fowler in his well-known catalog containing 72 refactorings for object-oriented code (Fowler and Beck 1999). However, the concept of *program transformations* has been discussed since the 1970s by Burstall and Darlington (1977). This term is equivalent to refactoring and commonly used in the context of functional languages (Hudak 1989; Partsch and Steinbrüggen 1983).

According to Thompson (2005), an important characteristic of refactorings is that they are diffuse. In other words, their effects are not restricted to a specific point in code, so it is necessary to consider the side effects of a refactoring during its application. An example of this diffusion can be seen in the application of the RENAMING refactoring, which is the most frequently refactoring performed by developers (Golubev et al. 2021). When we rename the definition of a function, we also need to update all the points in the code where that function is called, something that can spread across various files, classes, and modules of a software.

Many studies have been conducted in the field of refactoring, with the main goals of characterizing the granularity of refactorings (Brito et al. 2024), assessing their impact on software quality (Al Dallal and Abdin 2018; Bibiano et al. 2020; Ratzinger et al. 2008; Weißgerber and Diehl 2006), developing tools for detecting and/or automatically performing refactorings (Brown et al. 2011; Li et al. 2006; Lövei et al. 2008; Rowe et al. 2019b; Sagonas and Avgerinos 2009; Sherwany et al. 2015), comprehending the reasons why practitioners perform refactoring (Abid et al. 2020; Silva et al. 2016), and also investigate how and when refactorings are performed (Murphy-Hill et al. 2012; Oliveira et al. 2019; Xing and Stroulia 2006).

An example of a refactoring cataloged by Fowler and Beck (1999) is PULL UP METHOD. The Java code in Listing 2 illustrates a class hierarchy where `Bar` (Line 5) and `Qux` (Line 9) inherit from the class `Foo` (Line 1). These two subclasses evolved independently, resulting in the duplication of methods within the hierarchy, as seen in the `doSomething()` method definitions on Lines 6 and 10. In such cases, the PULL UP METHOD refactoring can be applied to eliminate duplicated code, thereby enhancing the codebase's reusability and maintainability. Listing 3 shows the refactored version, where the duplicated `doSomething()` method has been moved to the superclass `Foo` (Line 3). Since both `Bar` and `Qux` inherit the `doSomething()` method from `Foo`, the original behavior is preserved, as demonstrated in Lines 15 and 16.

Listing 2 Example of Java code before Pull up method

```java
public class Foo {
   ...
}

public class Bar extends Foo {
   public void doSomething() {...} // duplicated
      code
}

public class Qux extends Foo {
   public void doSomething() {...} // duplicated
      code
}

public class Main {
   public static void main(String[] args) {
      Foo bar = new Bar();
      Foo qux = new Qux();

      bar.doSomething();
      qux.doSomething();
   }
}
```

Listing 3 Example of Java code after Pull up method

```java
1  public class Foo {
2     ...
3     public void doSomething() {...}  // pulled up
         method
4  }
5
6  public class Bar extends Foo {...}
7
8  public class Qux extends Foo {...}
9
10 public class Main {
11    public static void main(String[] args) {
12       Foo bar = new Bar();
13       Foo qux = new Qux();
14
15       bar.doSomething();  // call Superclass
            method
16       qux.doSomething();  // call Superclass
            method
17    }
18 }
```

The Pull up method refactoring shown in Listing 3 is an example of a code transformation incompatible with Elixir. Unlike object-oriented languages such as Java, or even multi-paradigm languages like Kotlin, this functional language does not support inheritance. For the same reason, other refactorings cataloged by Fowler and Beck (1999), such as Push down method, Collapse hierarchy, and Extract superclass, are also not compatible with Elixir.

According to Bordignon and Silva (2020), an increasing number of developers are using functional languages in the industry. Conversely, the systematic literature review conducted by Abid et al. (2020) has shown the scarcity of studies conducted on refactoring for these languages. Furthermore, to the best of our knowledge, no study has yet investigated refactorings specifically tailored for Elixir. Therefore, this is a research opportunity that we are exploring in this paper.

3 Catalog of Refactorings for Elixir

In this section, we introduce the first study of this work, which centered on discovering and documenting refactorings compatible with Elixir, thereby suggesting a catalog. Although there are studies on refactoring in other functional languages such as Erlang (Li and Thompson 2006; Li et al. 2008; Lövei et al. 2008; Sagonas and Avgerinos 2009), Haskell (Brown et al. 2011; Lee 2011; Li and Thompson 2006; Li et al. 2005), OCaml (Rowe et al. 2019a, b), among others, to the best of our knowledge, only one preliminary study conducted by us Vegi and Valente (2023b) has sought so far to investigate and catalog specific refactoring strategies for Elixir. Therefore, in the present study, we aim to expand our preliminary short paper (Vegi and Valente 2023b) using a mixed methodological approach to propose a comprehensive catalog of refactorings for Elixir. Particularly, we want to answer the following research question:

***RQ1.* What are the refactoring strategies that occur in Elixir?** In this RQ, we seek to understand not only whether the 72 refactorings proposed in the nineties for object-oriented languages by Fowler and Beck (1999) are important in the Elixir context, but also to identify if other refactorings typical of functional languages or even specific refactorings occur in Elixir. In the present study, we identified 28 new refactoring strategies compatible with Elixir that were not cataloged in our preliminary short paper (Vegi and Valente 2023b). Out of these 28 refactorings, 14 belong to the ELIXIR- SPECIFIC group, which was nonexistent in our previous work.

We dedicate Section 3.1 to present the mixed methodology applied to proposing a catalog of refactorings for Elixir. Next, in Section 3.2, we present a catalog composed by 82 refactorings that are discussed and performed by Elixir developers. We have classified these refactorings into four groups (FUNCTIONAL, ELIXIR- SPECIFIC, ERLANG- SPECIFIC, and TRADITIONAL refactorings). Finally, Section 3.3 discusses threats to validity.

3.1 Study Design

Given that Elixir is a programming language created just 12 years ago and it is gaining recent popularity, only a limited number of studies have been conducted to explore software engineering aspects and quality attributes of code written in this functional language.

Nevertheless, since there are studies on refactorings for other functional languages, we decided to initiate our investigation by conducting a systematic literature review (SLR) to assess the compatibility of refactorings already studied in other functional languages with Elixir. Furthermore, our mixed methodological approach also included a grey literature review (GLR) and mining software repository (MSR) study. Figure 1 summarizes our steps to propose the catalog of refactorings for Elixir. We detail these steps in the following paragraphs and we will also highlight the differences between our present work and our preliminary study (Vegi and Valente 2023b).

1) Papers Search In this step, we begin a systematic literature review (SLR) to uncover refactorings compatible with code implemented in Elixir. To do this, we based on specific guidelines for conducting SLR studies in software engineering (Barn et al. 2017; Budgen

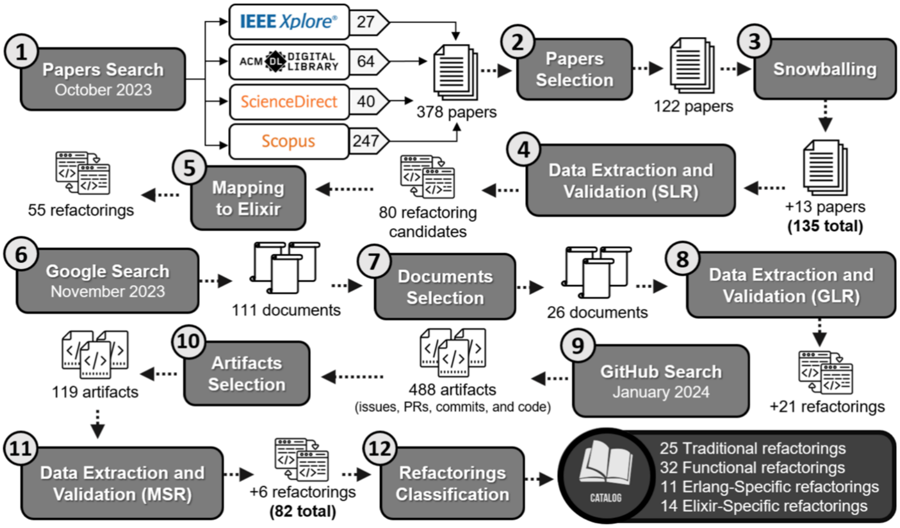


Fig. 1 Overview of methods for cataloging refactorings in Elixir

and Brereton 2006; Kitchenham and Charters 2007), which guided how we identified and analyzed scientific papers related to our research question. The decision to conduct a SLR to initiate the exploration of refactorings for Elixir stemmed from the existence of studies on refactorings for other functional languages, so it is natural to expect that some of these code transformations may also be useful for Elixir developers.

Therefore, at this step, our objective was to search for papers on refactoring in well-known functional programming languages. To construct a query string aligned with this objective, we followed some procedures proposed by Kitchenham and Charters (2007) to identify relevant search terms. Initially, we extracted the main terms using our research question as a foundation and also incorporated variations and synonyms of these main terms. Subsequently, to handle the variations related to the main terms, we utilized `OR` operators, and to combine the main terms themselves, we employed an `AND` operator. Following these procedures, we formulated the query presented in Listing 4.

Listing 4 SLR query string

```
("Refactoring" OR "Program Transformation") AND
   ("Functional Language" OR         "Functional
    Paradigm" OR "Alice" OR "APL" OR "Clojure"
   OR "Elixir" OR "Elm" OR "Erlang" OR "Haskell"
    OR "Julia" OR "Kotlin" OR "Lisp" OR "Logo"
   OR "ML" OR "Nim" OR "OCaml" OR "Racket" OR "
   SAS" OR "Scala" OR "Swift" OR "Wolfram")
```

The definition of the functional languages included in the query string presented in Listing 4 was carried out by analyzing the Top-100 most popular programming languages according to the TIOBE index.[10] The first author of this study examined these languages and selected only those classified as purely functional or as multi-paradigm with a strong influence of functional programming principles, such as data immutability. This selection

[10] https://www.tiobe.com/tiobe-index/

was subsequently validated by the second author, resulting in a comprehensive list of 22 functional languages.

After selecting these languages, we conducted preliminary searches to refine the query string, aiming to obtain more accurate results. During this calibration task, three out of 22 languages (*i.e.,* F#, Scheme, and J) were removed from the query string. Due to the characteristics of their names, their presence was polluting the results, returning many papers not related to our research goals. Moreover, in a preliminary inspection, we did not find any relevant paper when these languages were initially included.

In order to carry out the search, we opted for four reputable digital libraries that index software engineering publications relevant to our research questions: IEEE Xplore digital library,[11] ACM digital library,[12] ScienceDirect,[13] and Scopus.[14] We also adapted the query string to formats compatible with these digital libraries and the final search was conducted in October 2023. A total of 378 papers, labeled P1 to P378, were found after the search.[15]

2) Papers Selection The abstracts of all papers found in the previous step were reviewed by the first author, an expert Elixir developer, to identify only those pertinent to our research question. Of these, 288 papers were excluded because they are out of the context of our research. Furthermore, 90 other papers were discarded as duplicates, indicating their retrieval from multiple digital libraries. After this step, 122 papers were selected for full reading and detailed analysis. Our replication package encompasses all retrieved papers, including duplicates or those unrelated to our research question.

3) Snowballing After selecting the 122 papers in the previous step, we carried out a snowballing process to find other papers related to our research question that, for some reason, had not been found in the four digital libraries. To do this, we relied on recommendations by Wohlin (2014) for conducting snowballing in systematic reviews of software engineering studies. Therefore, the first author analyzed the titles of all papers referenced by the 122 papers selected in the previous step, selecting 13 new papers that are relevant to our research question. Subsequently, the abstracts of these papers were read, and all of them, referred to as P379 to P391, were selected for an in-depth analysis alongside the 122 papers selected in the previous step.

4) Data Extraction and Validation (SLR) All 135 papers selected in the previous steps were fully read by the first author to identify refactoring candidate strategies. In other words, refactoring candidates are those refactorings documented in the analyzed papers but whose compatibility with Elixir is not yet clear. In total, 106 refactoring candidates were extracted from 99 out of the 135 analyzed papers, with all these extractions being validated by the second author, as there were no disagreements between the authors. However, some of these candidates represented the same transformation but with different names. For example, as shown in Table 1, nine variations of the refactoring candidate RENAME AN IDENTIFIER were found.

11 http://ieeexplore.ieee.org

12 https://dl.acm.org

13 http://www.sciencedirect.com

14 https://www.scopus.com

15 In other words, our SLR identification system includes all 378 papers found in the search (step 1), along with 13 papers subsequently found through our snowballing process (step 3). All of these papers, labeled P1 to P391, can be consulted in the replication package of this study.

Table 1 Examples of variations of refactoring candidates

Variations of the refactoring Rename an identifier		
Rename a registered process	Rename variable	Rename function
Rename record	Rename record field	Rename macro
Rename header	Rename module	Rename directory

Considering the existence of these variations, the first author used thematic analysis, a technique for identifying patterns (or "themes") within a collection of data (Cruzes and Dybå 2011a, b; Silva et al. 2016) to group these candidates and then name the groups. At the end of this process, 80 different candidates were identified by the first author and validated by the second, with no disagreements. Among these 80 refactoring candidates, 69 did not have variations, while 11 resulted from grouping 37 variations of refactoring candidates, as can be seen in our replication package.

The quote extracted from the article P5 exemplifies our process of manually analyzing textual descriptions in the papers to prospect and validate refactorings for our catalog. As can be observed, it clearly describes a refactoring for Erlang code known as Register a process:

> *"[That refactoring registers] a process identifier with a user-provided name, and replaces the use of the process identifier in a `send` expression with the use of the process name whenever this is safe. Registering a process with a name allows any process in the system to communicate with the process without knowing its Pid [...]"*

In P38, we extracted a clear description of another refactoring characteristic of code implemented in Erlang. Since the use of traditional defensive programming mechanisms (*i.e.,* `try..catch`) is not idiomatic in this functional language (Li and Thompson 2006), the refactoring Moving error- handling mechanisms to supervision trees can be used to improve the design of Erlang code:

> *"[...] While defensive-style programming is a good programming practice when a sequential programming language is used, non-defensive style programming is the right thing to do when programming with Erlang [...] This refactoring helps to transform defensive-style sequential error-handling code written in Erlang into concurrent error-handling, typically using supervisor trees."*

As illustrated by the quotes extracted from P5 and P38, our process of manually analyzing textual descriptions aimed to identify refactoring candidates clearly described in the papers. The verbatim sources of the extractions performed for each of the 80 refactoring candidates identified in this step are listed in our replication package (Vegi and Valente 2024).

5) Mapping to Elixir In this step, all 80 refactoring candidates extracted from our SLR were analyzed by the first author to identify those compatible with Elixir's features and then adapt them to the language's syntax and semantics. Due to incompatibilities with some language features, 25 out of the 80 initial refactoring candidates were not selected to be included in our catalog of refactorings for Elixir. For example, 11 out of the 25 unselected refactoring are transformations that rely on Haskell's static data typing system, something that cannot be replicated in Elixir. Other six candidates were considered incompatible with Elixir because they rely on specific Erlang libraries, such as Skel[16] and SD Erlang,[17] that do not have implementations in Elixir.

Therefore, 55 out of the 80 refactoring candidates were classified as compatible with Elixir. Since they were originally extracted from other languages, the first author of this study produced code examples in Elixir to illustrate and document all these code transformation strategies. The examples were created by translating the existing code from the analyzed papers and, primarily, by interpreting the original descriptions of the refactorings extracted in the previous step. Additionally, we provide detailed documentation of the side conditions for Elixir refactorings, outlining the requirements that must be met before and after the transformations to ensure the behavior of the refactored code remains unchanged.

6) Google Search In this step, we began reviewing the grey literature (GLR)—which comprises documents not peer-reviewed, such as blogs, videos, forums, books, podcasts, etc.—aiming to find specific refactoring content for Elixir, something that was not possible in our SLR. This is a complementary methodological step commonly used when extracting knowledge on a topic that is still relatively unexplored in scientific articles (Kamei et al. 2021; Vegi and Valente 2022, 2023; Zhang et al. 2020), as is the case with refactorings for Elixir.

To define the query string used for conducting searches on Google in our GLR, we followed the guidelines proposed by Garousi et al. (2019). Specifically, we identified and combined keywords related to refactoring in Elixir. After that, we conducted preliminary experiments to calibrate the query string, ensuring that the combined terms and synonyms yield results aligned with the objectives of our research. Listing 5 presents our query after this calibration.

Listing 5 GLR query string

```
("Refactoring" OR "Program Transformation" OR "
   Refactorings" OR "Program Transformations")
   AND ("Elixir")
```

The main difference between this query and the one used in our SLR (Listing 4) lies in the reduction of variations for the second main term. In the SLR, it was known that there were no specific studies on refactoring for Elixir. Therefore, we combined the main term "`Refactoring`" with variations of the term "`Functional Language`" (*i.e.,* a list of 19 functional languages). However, in the GLR, considering that our key goal is to complement the SLR study with refactorings that focus specifically on Elixir, we chose to replace the main term "`Functional Language`" and its variations with just "`Elixir`". Additionally, we used both singular and plural terms. This approach was taken to mitigate the risk that some discussions would not be retrieved by Google. The final search was carried out in November 2023.

7) Documents Selection After running our query string on Google, 111 documents were returned, which we refer to as G1 to G111. Considering that PageRank always sorts Google's search results in descending order of relevance (Page et al. 1999), the tendency is that documents farther from the top positions in this ranking have more potential to be out of context. Taking into account this characteristic of Google's search algorithm, as done in our previous

[16] https://github.com/ParaPhrase/skel

[17] https://www.dcs.gla.ac.uk/research/sd-erlang/

Table 2 Distribution of valid documents among authority criteria

Authority criteria	# Documents
(a) is the publisher a company that works with Elixir?	2
(b) is the author associated with a company that works with Elixir?	18
(c) has the author published other works in the field?	6

study on code smells for Elixir (Vegi and Valente 2022, 2023), we established a stopping criterion to avoid analyzing contextless documents. This criterion states that when four consecutive pages with less than 50% of valid documents were found, we would stop selecting documents. However, the valid documents in these four pages will not be discarded (Vegi and Valente 2022, 2023). This requirement to limit the number of documents to be analyzed is recurring in GLRs, as described by Garousi et al. (2019).

We also replicated the validity criteria used in our previous study (Vegi and Valente 2023), which is an adaptation of the Quality Assessment Checklist proposed by Garousi et al. (2019). Therefore, the documents were analyzed by the two authors of this work in the order returned by Google, and to be considered valid, first, the documents should not be authored by the authors of this work. Second, they should be related to the context of our research question (RQ1). Third, they should meet at least one of the following authority criteria: (a) is the publisher a company that works with Elixir?[18] (b) is the author associated with a company that works with Elixir? (c) has the author published other works in the field?

After analyzing the first seven pages of returned documents, our stopping criterion was reached, establishing a saturation point. Among the 70 documents analyzed until reaching the stopping criterion, 26 of them were selected as valid, while the other 44 were discarded. Out of these 44 discarded, 11 are from the authors of this work,[19] 31 were not related to our RQ, and two did not meet the authority criteria. Table 2 shows the distribution of the 26 valid documents among the three authority criteria.

We checked the authors' profiles on the platforms where the documents were published to ensure compliance with the proposed authority criteria. The representativeness of the selected documents varies from individual publications on blogs or forums like Elixir Forum and StackOverflow, made by highly active members in Elixir-related topics, to developers who have worked in companies using Elixir, such as Remote,[20] Veeps,[21] Trybe,[22] Plataformatec,[23] among others. Additionally, documents from the official Elixir documentation or authored by companies working with Elixir, such as DockYard,[24] were also selected.

8) Data Extraction and Validation (GLR) All 26 documents selected in the previous step were read and analyzed in detail by the first author to find discussions about refactoring strategies performed in Elixir. Subsequently, the second author validated the decisions made regarding extracting or not refactoring strategies from these documents, with no disagree-

[18] Companies using Elixir in production code: https://elixir-companies.com

[19] Example [G3]: https://elixirforum.com/t/towards-a-catalog-of-refactorings-for-elixir/

[20] https://remote.com/

[21] https://veeps.com/

[22] https://betrybe.com/

[23] https://plataformatec.com/

[24] https://dockyard.com/

ments. We discarded only two out of 26 documents—G24 and G39—because they describe very specific code transformations and therefore could not be generalized as refactoring strategies applicable to different scenarios.

An example of a valid document that had no refactoring extracted can be seen in G24. In that document, posted on a personal blog by an experienced Elixir developer, there is an indication of a possible refactoring strategy that involves substituting conditional statements with a mathematical approach:

> *"[Do you need trying to refactor] and improve the game [...] I found a mathematical approach to solving the logic of our game. [...] Now we use modular arithmetic to add a few math to our code and make the code brighter and cleaner."*

Although in the context discussed in G24 this transformation has resulted in the reduction of many lines of code and quality improvements, we consider that the approach used, based on the remainder of an integer division, is too specific to the rules of the game *Rock-Paper-Scissors*, therefore not recommended to be included in a catalog. On the other hand, through our process of manually analyzing textual descriptions and code, we extracted from document G70 a clear description of the refactoring PIPELINE USING "WITH", which is specific to code implemented in Elixir:

> *"[...] The `with` Elixir's statement was designed to help you with a chain of conditional operations. We have the Elixir's pipe operators (|>) that works well with pure functions or stable returns. But, when we need to deal with conditional returns, the pipe version will require extra functions to branch the code. [Therefore] The `with` version is shorter, easier to read, and doesn't have extra unnecessary functions. [...]"*

Initially, 41 refactorings were extracted in this step. However, as happened in the SLR, some of these refactorings actually represented the same code transformation, but with different names. To group these variations, we conducted again a thematic analysis (Cruzes and Dybå 2011a, b; Silva et al. 2016), as described in step 4. At the end, we found 35 different refactorings, with 21 of them being new compared to previous steps. Only one out of these 35 refactorings—REPLACE PIPELINE WITH A FUNCTION—resulted from the grouping of seven variations.

9) GitHub Search In this step, we conducted a study of mining software repositories (MSR) on GitHub, aiming to expand our catalog of refactorings through the analysis of artifacts directly related to code implemented in Elixir. To mine these repositories, we followed the two main steps suggested by Dabic et al. (2021) for conducting this type of study, which are first to define a criterion for selecting the repositories and subsequently to define a query string related to the research question (RQ1).

We selected the Top-10 Elixir repositories with the most stars on GitHub and then mined artifacts only from them. With this decision, our goal was to analyze only well-engineered projects and, therefore, reduce the noise during the analysis. In this selection, repositories that contain only documentation (*e.g.,* Awesome Elixir[25]) were disregarded. Table 3 lists the repositories that were the subject of study in our MSR. The official Elixir repository (*i.e.,* `elixir-lang/elixir`) returned 135 artifacts in our searches (step 9), the highest number among all 10 repositories. Furthermore, it was also the repository with the largest

[25] https://github.com/h4cc/awesome-elixir

Table 3 Top-10 Elixir repositories with the most stars

Repository	Stars	Retrieved*	Selected*
https://github.com/elixir-lang/elixir	22.5k	135	43
https://github.com/phoenixframework/phoenix	20.3k	30	9
https://github.com/plausible/analytics	17.4k	44	14
https://github.com/supabase/realtime	6.3k	9	4
https://github.com/elixir-ecto/ecto	5.9k	56	15
https://github.com/firezone/firezone	5.9k	75	8
https://github.com/phoenixframework/phoenix_live_view	5.6k	17	1
https://github.com/papercups-io/papercups	5.5k	28	6
https://github.com/teslamate-org/teslamate	4.9k	8	0
https://github.com/rrrene/credo	4.8k	86	19
	Total	488	119

* Number of Artifacts

number of valid and therefore selected artifacts, according to the selection criteria defined in step 10 (43 out of 135).

To find artifacts—ISSUES, PULL REQUESTS, COMMITS, and FILES—that contain references to refactorings, we used a query string similar to the one used in our GLR (Listing 5), as shown in Listing 6.

Listing 6 MSR query string

```
"refactoring" OR "program transformation" repo:
    user/repository
```

We executed this query in January 2024 for each of the 10 selected repositories (Table 3), replacing the `user/repository` part with the data from each respective repository (*e.g.*, `elixir-lang/elixir`). Moreover, we used the standard GitHub search service without any additional qualifiers.

10) Artifacts Selection As shown in Tables 3 and 4, a total of 488 artifacts, which we refer to as M1 to M488, were retrieved in the searches conducted in the previous step. Out of these 488 artifacts, 119 were selected to be further analyzed in detail.

Table 4 Overview of artifacts selection

Artifact	Retrieved	Selected
SOURCE CODE FILES	25	3
COMMITS	118	31
ISSUES	106	26
PULL REQUESTS	239	59
Total	488	119

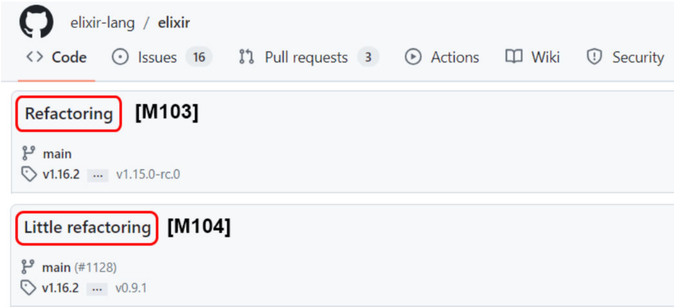


Fig. 2 Examples of unclear commits

To select these 119 artifacts, all 488 retrieved artifacts were analyzed by the first author and subsequently validated by the second author, aiming for agreement on the fulfillment of the following selection criteria. First, to be selected, an artifact should align with the context of our research question (RQ1). Second, the artifact cannot have been previously retrieved in our GLR or authored by the authors of this study.

Particularly, the inspection of COMMITS and PULL REQUESTS did not involve code analysis but only their titles and textual descriptions. In this way, 52 of these artifacts were discarded for containing ambiguous messages or not precisely describing the performed code transformation. Figure 2 shows an example with two COMMITS—M103 and M104—discarded due to lack of clarity. The titles of these commits just mention the words "refactoring" or "little refactoring". Furthermore, the commit descriptions do not provide information about the refactorings that were performed.

Among the other 317 discarded artifacts, 299 are not related to the context of our RQ, and 18 are co-authored by the authors of this work.[26] Table 4 provides an overview of the retrieved and selected artifacts organized by types. The most retrieved and selected artifact type was the PULL REQUEST. Out of the 239 PULL REQUESTS found in our searches (step 9), 59 were considered valid and therefore were selected according to the criteria defined in step 10 (*i.e.,* 24.7%). In contrast, the selection rate for SOURCE CODE FILES was only 12% (three out of 25 artifacts).

11) Data Extraction and Validation (MSR) In order to extract sentences, comments, or changes that characterize instances of refactorings among the 119 artifacts selected in the previous step, the first author of this work read and analyzed these artifacts thoroughly. Unlike the process used to select artifacts, for extracting the refactorings, the first author not only analyzed the textual descriptions but also examined the code diffs to understand better the details of the changes made in the codebase.

All extractions were later validated by the second author, seeking to reach an agreement on whether they represent a refactoring or not. The authors agreed on all the extractions, and only three artifacts out of 119 analyzed—M9, M15, and M165—did not have any refactorings extracted. An example of an artifact with no refactoring can be seen in M9. In this ISSUE, a developer describes a code transformation related to vertical alignment for multiline `keyword lists`,[27] code comments, and pipe indentation in Elixir:

[26] Example [M38]: https://github.com/elixir-lang/elixir/pull/12952

[27] https://hexdocs.pm/elixir/keywords-and-maps.html#keyword-lists

> *"[I want to refactor the code] to make multiline keyword lists easier to read by aligning the values into a column. [Without whitespaces] within the lines it is way harder to read."*

Although this description indicates a transformation that can improve code readability, we understand that issues related to formatting style can be opinionated, as evidenced by the discussions generated in M9, where divergent comments were made by other developers. For this reason, it was discarded. On the other hand, in M212, we extracted a textual description that allowed us to catalog the refactoring REPLACE "ENUM" COLLECTIONS WITH "STREAM". During the code review process for this PULL REQUEST, a developer clearly suggested this refactoring by making the following comment:

> *"I would suggest to [refactor the code using] the* `Stream` *module instead of the* `Enum` *module if multiple operations like these are used together. This can lower the times you traverse the list, while keeping the same idea. [...]"*

In total, 179 instances of 46 different refactorings were extracted in this mining study. Among the 46 refactorings extracted, six are new compared to previous steps, thus completing the catalog with a total of 82 refactorings.

12) Refactorings Classification Finally, we analyzed all the refactorings in the catalog and categorized them into four groups. This classification was done based on the programming features used by the code transformations, as described in Table 5.

Although all 82 refactorings in our catalog are compatible with Elixir (see step 5), we created four categories, presented in Table 5, to organize them considering their potential for generalization. Refactorings in the TRADITIONAL category are the most generic, as they are not only compatible with Elixir but also potentially applicable to a wide range of languages, including object-oriented ones. In contrast, refactorings in the FUNCTIONAL category are specific to languages rooted in the functional paradigm, such as Elixir, Erlang, and others. Furthermore, it is important to mention that Erlang provides a robust set of libraries, collectively known as OTP, which simplify the implementation of distributed and scalable systems across its entire ecosystem (Jurić 2024). As explained in Section 2.1, Erlang code is interoperable

Table 5 Categories used to organize the catalog of refactorings for Elixir

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

with Elixir since both languages are interpreted by the same virtual machine (BEAM), meaning Elixir is part of the broader Erlang ecosystem, which also includes languages like Gleam.[28] Thus, the ERLANG- SPECIFIC category encompasses all code transformations related to features shared by the Erlang ecosystem, including Elixir and other BEAM-based languages. Finally, the ELIXIR- SPECIFIC category, our least generic one, contains transformations that have the lowest potential for compatibility with other languages, as they directly involve features and libraries created exclusively for Elixir.

3.2 What are the Refactoring Strategies that Occur in Elixir? (RQ1)

Using our mixed methodological approach, we cataloged 82 refactorings for Elixir found in discussions contained in 239 different sources. In the following subsections, we describe the refactorings from the categories FUNCTIONAL, ELIXIR- SPECIFIC, ERLANG- SPECIFIC, and TRADITIONAL, respectively. A subset of these refactorings was chosen by us to offer more detailed explanations in these subsections. This selection was made because they collectively provide an overview of the primary features of their respective categories, thus enabling a good comprehension of the entire catalog.

A comprehensive documentation of all 82 cataloged refactorings is available in our GitHub repository: https://github.com/lucasvegi/Elixir-Refactorings, where each refactoring is presented in a structured format, including its motivations, mechanics, side conditions, and illustrative code examples before and after the transformation. This repository serves as a concrete representation of our catalog, offering quick access to detailed metadata for each refactoring strategy.

3.2.1 Functional Refactorings

This category had the highest number of Elixir-compatible refactorings cataloged, with 32 (out of 82). Therefore, to improve the readability of our paper, we chose to present in Tables 6 and 7 only those found in at least three different sources. Consequently, the descriptions of the remaining 10 refactorings from this category can be found in Table 17 (Appendix).

Pattern matching is a mechanism that checks whether a sequence of tokens follows a specific pattern. It is often used in functional programming languages to destructure complex data and perform different actions based on the patterns found in expressions (Scott 2015). In Elixir, pattern matching can be useful for assigning variables, unpacking values, and acting as a control-flow mechanism (Almeida 2015). The refactoring INTRODUCE PATTERN MATCHING OVER A PARAMETER was found in 23 sources, being one of the most extracted in this category. It can be used to replace conditional structures (*e.g.,* `if`, `cond`, and `case`) that control the branches defined by a function's parameter, as described by the author of G16:

> *"[Let's replace] the `case` statement conditional to pattern matching [...] Since Elixir and Erlang let you define multiple function heads for functions with the same arity [multi-clause function], we can rely on pattern matching on our arguments and move more specific, conditional cases towards the top, leaving default and generic cases as the last definitions [...]"*

[28] https://gleam.run/

Table 6 Functional Refactorings compatible with Elixir - Part 1

Refactoring	Description	#
Generalise a function definition	Creates a new `higher-order function` to generalize different `functions` that have equivalent expressions	23
Introduce pattern matching over a parameter	Replaces conditional structures (*e.g.,* `if`, `unless`, `cond`, `case`) that control the branches of a function, with pattern-matching and `multi-clause functions`	23
Turning anonymous into local functions	Transforms identical `anonymous functions` defined at different places (*i.e.,* duplicated code) into a single `local function`—*a.k.a. lambda lifting*	11
Nested list functions to comprehension	Transforms nested calls to `Enum.map/2` and `Enum.filter/2` into a list comprehension (*i.e.,* `for` construct). This transformation avoids multiple traversals over a data structure and intermediate lists—*a.k.a. deforestation*	10
Replace pipeline with a function	Replaces a pipeline composed of Elixir's built-in higher-order functions (*e.g.,* `Enum.map/2 |> Enum.into/2`) with a call to one of these functions or by calling another built-in function with equivalent behavior (*e.g.,* `Enum.into/3`)	10
From tuple to struct	Transforms `tuples` into `structs`, which are data structures that allow naming their fields, thus hiding some details of the data representation	9
Merging multiple definitions	Groups complementary `functions`, that have identical code snippets, into a single `function` that returns a `tuple`. Each original return provided by the merged `functions` will be contained in different elements of the `tuple` returned by the new `function`—*a.k.a. tupling*	8

#: Number of sources

Listing 7 shows an example of code where Introduce pattern matching over a parameter can be used. The `fibonacci/1` function has three different branches defined by the value of its single parameter: two for its base cases (Lines 5 and 6) and one for its recursive case (Line 7). The control flow for these branches is managed by a `case` statement (Line 4). This refactoring can be used to replace the `case` statement by a pattern matching over the parameter `n`, as shown in Listing 8. By performing this refactoring, `fibonacci/1` transforms into an Elixir multi-clause function, which is a group of functions with the same name, where the first two clauses handle the base cases (Lines 3 and 4), and the last clause handles the recursive case (Line 5), thus preserving the behavior of the code.

Listing 7 Example of a code before Introduce pattern matching over a parameter

```
1  # Before refactoring:
2
3  def fibonacci(n) when is_integer(n) do
4    case n do
5      0 -> 0
6      1 -> 1
7      _ -> fibonacci(n-1) + fibonacci(n-2)
8    end
9  end
```

Table 7 Functional Refactorings compatible with Elixir - Part 2

Refactoring	Description	#
TRANSFORM A BODY- RECURSIVE FUNCTION TO A TAIL- RECURSIVE	Converts a body-recursive `function` into a tail-recursive one, thus improving runtime performance due to the BEAM's tail-call optimization	8
TRANSFORM TO LIST COMPREHENSION	Transforms calls to `Enum.map/2` or `Enum.filter/2` into list comprehensions, thus creating a semantically equivalent code	8
REMOVE SINGLE PIPE	Replace a pipe that doesn't involve multiple chained `function` calls (*i.e.,* those that have only two members, with the first being a variable or a zero-arity `function`, followed by a non-zero-arity `function` call), with a call to the `function` that was originally the last pipe member	7
CLOSURE CONVERSION	Transforms closures (*i.e.,* anonymous `functions` that access variables outside their scope) into `functions` that receive those variables as parameters	6
EQUALITY GUARD TO PATTERN MATCHING	Replaces a temporary variable extracted from a `struct` field, that is only used in an equality comparison in a guard, with pattern matching	6
IMPROVING LIST APPENDING PERFORMANCE	Replaces tail concatenations into a `list` with head concatenations, thus increasing the amount of shared memory between the intermediate lists	6
INTRODUCE ENUM.MAP/2	Replaces a list expression in which each element is generated by calling the same `function` with a call to the higher-order function `Enum.map/2`	6
SPLITTING A DEFINITION	Separates a recursive `function` by creating distinct recursive `functions`, each responsible for individually generating a respective element originally contained in a `tuple`	6
STRUCT FIELD ACCESS ELIMINATION	Replaces direct access to fields of a `struct` with temporary variables that hold values extracted from these fields	5
CONVERTS GUARDS TO CONDITIONALS	Replaces all guards in a multi-clause `function` with traditional conditionals (*e.g.,* `if` or `cond`), creating only one clause for the `function`	4
WIDEN OR NARROW DEFINITION SCOPE	Widens the scope of an anonymous `function` defined inside a named `function`, by transforming it into a new named `function`. The reverse operation can also narrow the scope	4
ELIMINATE SINGLE BRANCH	Simplifies the code by eliminating control statements that have only one possible flow	3
FUNCTION CLAUSES TO/FROM CASE CLAUSES	Transforms a multi-clause `function` into a single-clause one, mapping `function` clauses into clauses of a `case` statement	3

Table 7 continued

Refactoring	Description	#
INLINE MACRO	Replaces a `macro` with the code defined in its body. It can be useful when a `macro` is created to solve problems that `functions` or other pre-existing Elixir structures could solve	3
REPLACE FUNCTION CALL WITH RAW VALUE IN A PIPELINE START	Changes the beginning of a pipeline (*i.e.,* sequence of |>), extracting the initial parameter from the `function` call that originally starts the pipe and incorporating this value at the pipeline's start	3

#: Number of sources

Listing 8 Example of a code after Introduce pattern matching over a parameter

```
# After refactoring:

def fibonacci(0), do: 0
def fibonacci(1), do: 1
def fibonacci(n) when is_integer(n) do
  fibonacci(n-1) + fibonacci(n-2)
end
```

GENERALISE A FUNCTION DEFINITION also emerged from 23 sources, making it the most extracted FUNCTIONAL refactoring in our SLR. This refactoring aims to use higher-order functions—which are functions that take one or more functions as parameters—to eliminate duplicated code among functions with equivalent expressions. To achieve this, these functions can be generalized into a new one, which is later called within the bodies of the redundant functions. Listing 9 illustrates an opportunity to apply this refactoring. Although `foo/1` and `bar/1` transform lists differently, they share equivalent expressions. The `foo/1` changes a list in two stages: first, it squares each element of the list (Line 4), and then it multiplies each of them by three (Line 5), resulting in a new list. Likewise, `bar/1` takes a list, doubles the value of each element (Line 9), and then creates a new list containing only those multiples of four (Line 10).

Listing 10 shows the result of performing this refactoring. Since `foo/1` and `bar/1` have equivalent expressions, this refactoring generalizes both by creating a new function `generic/4` (Line 3). Additionally, the bodies of `foo/1` and `bar/1` are replaced with calls to `generic/4` (Lines 9 and 13). Note that `generic/4` is a higher-order function because its last three parameters are anonymous functions that are called only within its body (Lines 4 and 5).

Listing 9 Example of a code before Generalise a function definition

```
1  # Before refactoring
2
3  def foo(list) do
4    list_comprehension = for x <- list, do: x * x
5    Enum.map(list_comprehension, &(&1 * 3))
6  end
7
8  def bar(list) do
9    list_comprehension = for x <- list, do: x + x
10   Enum.filter(list_comprehension, &(rem(&1, 4)
       == 0))
11 end
```

Listing 10 Example of a code after Generalise a function definition

```
1  # After refactoring
2
3  def generic(list, generator_op, trans_op, trans_args) do
4    list_comprehension = for x <- list, do: generator_op.(x, x)
5    trans_op.(list_comprehension, trans_args)
6  end
7
8  def foo(list) do
9    generic(list, &(&1 * &1), &Enum.map/2, &(&1 * 3))
10 end
11
12 def bar(list) do
13   generic(list, &(&1 + &1), &Enum.filter/2, &(rem(&1, 4) == 0))
14 end
```

A list comprehension is a syntactic construct capable of creating a list based on existing ones. This feature is inspired by the mathematical notation for defining sets and is very common in functional languages like Elixir.[29] List Comprehension Simplifications is a refactoring technique that can be used to transform an Elixir list comprehension (*i.e.,* `for` construct) into semantically equivalent calls to the functions `Enum.map/2` or `Enum.filter/2`, thus improving the code readability by facilitating the visualization of chains of functional transformations.

The function `generic/4`, shown in the code refactored by Generalise a Function Definition (Listing 10), therefore presents an opportunity for applying the refactoring List Comprehension Simplifications, since a list comprehension is used to perform the first of the two transformation steps (Line 4). Listing 11 shows the result of performing this refactoring, where the existing list comprehension was replaced by a call to the function `Enum.map/2` (Line 4). Furthermore, since the temporary variable `list_comprehension` is no longer needed to store an intermediate list, it was removed (see Temporary Variable Elimination in Table 10), and a pipe operator was used to pass the value returned by `Enum.map/2` as the first parameter of the anonymous function `trans_op` (Line 5).

Listing 11 Example of a code after List comprehension simplifications

```
1  # After refactoring
2
3  def generic(list, generator_op, trans_op, trans_args) do
4    Enum.map(list, generator_op)
5    |> trans_op.(trans_args)
6  end
```

In Elixir, as well as in other functional languages like Erlang and Haskell, functions are considered first-class citizens. This means they can be assigned to variables, allowing for the creation of *anonymous functions*, also known as *lambda* (Jurić 2024). Although *anonymous functions* are useful, they have less potential for reuse than local functions and cannot be exported to other modules, for example. Turning Anonymous into Local Functions was found in 11 sources, ranking as the third most extracted Functional refactoring. This refactoring strategy aims to transform identical *anonymous functions* defined in different parts of the codebase into a single *local function*. Additionally, the locations where the

29 https://hexdocs.pm/elixir/comprehensions.html

anonymous functions were originally implemented are updated to use the new *local function*. This refactoring is also known as *lambda lifting* (Sagonas and Avgerinos 2009), as described by the authors of the paper P6:

> *"This transformation is known as lambda lifting in functional languages [...] It transforms a [lambda] to local function and changes the point where the [lambda] was previously applied to a function call [...]"*

Listing 12 shows two different local functions—`foo/1` and `bar/1`—defining the same lambda `fn x -> x * 2 end` (Lines 4 and 8), resulting in duplicated code. As shown in Listing 13, after performing Turning anonymous into local functions, the lambda `fn x -> x * 2 end` was transformed into the local function `double/1` (Line 3), and the places where this lambda was defined were updated to use `double/1` (Lines 8 and 12). This transformation led to improvements in code maintainability and reusability. Instead of having scattered identical anonymous functions throughout the codebase, which can be harmful when code changes are needed, we extracted and concentrated this duplicated code in just one place—`double/1`.

Listing 12 Example of a code before Turning anonymous into local functions

```
1  # Before refactoring
2
3  def foo(list) do
4    Enum.map(list, fn x -> x * 2 end)
5  end
6
7  def bar(list) do
8    Enum.map_every(list, 3, fn x -> x * 2 end)
9  end
```

Listing 13 Example of a code after Turning anonymous into local functions

```
1  # After refactoring
2
3  def double(x) do
4    x * 2
5  end
6
7  def foo(list) do
8    Enum.map(list, &double/1)
9  end
10
11 def bar(list) do
12   Enum.map_every(list, 3, &double/1)
13 end
```

Convert nested conditionals to pipeline is a composite refactoring found in two sources from our GLR and MSR studies. This type of refactoring, as described by Brito et al. (2024), is a code transformation of larger granularity, characterized by being a sequence of primitive refactorings, such as the Traditional refactorings proposed in our catalog. This Functional refactoring aims to eliminate nested conditionals used only to control a sequence of function calls, replacing them with pipe operators. The interfaces of functions involved in the pipeline are also modified by adding parameters and using pattern matching.

Listing 14 exemplifies an opportunity to apply this refactoring. The function `update_game_state/3` uses nested conditional statements—`if` (Line 5) and `case` (Line 7)—to ensure the safe invocation of the next function in the sequence: `valid_move/2` (Line 4), `players_turn/2` (Line 6), and `play_turn/3` (Line 8).

Listing 14 Example of a code before Convert nested conditionals to pipeline

```
# Before refactoring:

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

Listing 15 Example of a code after Convert nested conditionals to pipeline

```
1 # After refactoring:
2
3 defp update_game_state(%{status: :started} =
      state, index, user_id) do
4   state
5   |> valid_move(index)
6   |> players_turn(state, user_id)
7   |> play_turn(state, index, marker)
8 end
```

Listing 15 shows the result of applying Convert nested conditionals to pipeline. The refactored code continues to ensure the safe invocation of the next function in the sequence, as in the previous version, but it reduces the number of lines of code and improves readability. To maintain the same code behavior, this refactoring increased the arity of `players_turn/2` and `play_turn/3`, transforming them into `players_turn/3` (Line 6) and `play_turn/4` (Line 7), respectively. The additional parameter in each of these functions is meant to receive the returns of the previous functions in the pipeline—which are in the Elixir's patterns `{:ok, _}` or `{:error, _}`—and then guides their internal flows. Both Listings 14 and 15 are based on an original code provided by Gary Rennie in G14.

> **Finding #1:** Through three distinct research methods, we found 32 functional refactorings compatible with Elixir. This category holds the highest number of refactorings in the catalog, comprising 39% of them.

3.2.2 Elixir-Specific Refactorings

To the best of our knowledge, this is the first study that catalogs Elixir- Specific refactorings. Unlike the other 68 refactorings that compose our catalog, the 14 refactorings in this category perform code transformations that depend on programming features unique to this language, and therefore are not adaptations of refactorings initially proposed for other contexts. Due to these factors, in this section, we chose to present and describe all the Elixir- Specific refactorings in Table 8.

Table 8 Elixir-Specific Refactorings

Refactoring	Description	#
PIPELINE USING "WITH"	Replaces the nested use of conditional statements to control a sequence of `function` calls, with a kind of pipeline using a `with` statement	7
ALIAS EXPANSION	Expands multi-alias instructions fused into one (*e.g.,* `alias Foo.Bar.{Baz, Boom}`), transforming them into separate alias instructions (*e.g.,* `alias Foo.Bar.Baz` and `alias Foo.Bar.Boom`)	5
TRANSFORM NESTED "IF" STATEMENTS INTO A "COND"	Transforms multiple nested `if` statements, used to compensate for the absence of the `else if` construct in Elixir, into a `cond` statement	5
EXPLICIT A DOUBLE BOOLEAN NEGATION	This refactoring replaces a double boolean negation with a new helper `multi-clause function`	2
MOVING "WITH" CLAUSES WITHOUT PATTERN MATCHING	A `with` statement can be defined using an initial or final clause without pattern matching. This refactoring can: (1) move outside an initial clause that doesn't match anything, placing it just before the `with`; or (2) move into the body of a `with` a final clause that doesn't match anything	2
PIPELINE FOR DATABASE TRANSACTIONS	Converts an `anonymous function` used in a `Ecto.Repo.transaction/2` call, into an `Ecto.Multi` instance	2
TRANSFORM "IF" STATEMENTS USING PATTERN MATCHING INTO A "CASE"	This refactoring transforms an `if` statement that uses pattern matching, into a `case` conditional statement	2
DEFAULT VALUE FOR AN ABSENT KEY IN A MAP	Replaces a `Map.has_key?/2` call together an `if...else`, with only a `Map.get/3` call. This can be used when we expect a `Map` to have a certain key, and if not, we need to provide a default value	1
DEFINING A SUBSET OF A MAP	Replaces the manual creation of a `Map` subset, performed by accessing individually each of the desired key/value pairs, with a call to `Map.take/2`	1
GENERALISE A PROCESS ABSTRACTION	Transforms `Task` or `Agent` process abstractions into `GenServer`	1
MODIFYING KEYS IN A MAP	Replaces the combined usage of `Map.get/2`, `Map.put/2`, and `Map.delete/2` with a `Map.new/2` call and a multi-clause `lambda`, so changing a `Map`'s key name, but keeping it pointing to the same value	1
REMOVE REDUNDANT LAST CLAUSE IN "WITH"	This refactoring removes a redundant last clause in `with` and replaces the predefined value to be returned by this statement with the expression that was checked in the redundant clause	1

Table 8 continued

Refactoring	Description	#
Replace "Enum" collections with "Stream"	Replaces the use of the `Enum` module with the `Stream` module when multiple operations in large collections are performed in a pipeline	1
Simplifying Ecto schema fields validation	Replaces a list of `Ecto` schema fields created manually, used for validations (*i.e.,* `validate_required/3`), with a call to the `Ecto __schema__/1`	1

#: Number of sources

The `with` statement is an Elixir-specific conditional statement. This conditional is used for pattern matching chaining. It compares the results of several expressions with patterns, returning a predefined value if all patterns match, or the result of the first expression that does not match a pattern (Jurić 2024). Listing 16 presents an example of using a `with` statement containing two clauses (Lines 1 and 2), where the results of `expression_1` and `expression_2` are respectively compared with `pattern_1` and `pattern_2`. If the results of these two expressions match their respective patterns, the `with` statement will return `predefined_value` (Line 3); otherwise, it will return the result of the expression that did not match.

Listing 16 Example of a "with" conditional statement

```
1  with pattern_1 <- expression_1,
2       pattern_2 <- expression_2 do
3    predefined_value
4  end
```

Pipeline using "with" is a transformation that depends on the `with` statement. When conditional statements, such as `if..else` and `case`, are nested
to control sequences of function calls, the code's readability can become compromised. In these situations, we can replace nested conditionals with a kind pipeline using a `with` statement, thus performing pattern matching at each function call and interrupting the pipeline if any pattern does not match.

As already shown in Listing 14, the function `update_game_state/3` uses nested conditional statements to control a sequence of calls to `valid_move/2`, `players_turn/2`, and `play_turn/3`. Although `update_game_state/3` can be refactored using Convert nested conditionals to pipeline, as presented in Listing 15, this function can also be refactored using Pipeline using "with", as these two strategies can be applied in the same opportunities.

Listing 17 shows the result of refactoring `update_game_state/3` using Pipeline using "with". This example is based on the original code by Gary Rennie, provided in G14. Using this transformation, the calls to `valid_move/2` (Line 4), `players_turn/2` (Line 5), and `play_turn/3` (Line 6) were chained using pattern matching in each of the three clauses of a `with` statement, thus eliminating the need to use nested conditional statements. Additionally, unlike the transformation using Convert nested conditionals to pipeline (Listing 15), Pipeline using "with" does not require modifications in the interfaces of the functions involved in the transformation.

Listing 17 Example of a code after Pipeline using "with"

```
1  # After refactoring:
2
3  defp update_game_state(%{status: :started} =
      state, index, user_id) do
4    with {:ok, _}            <- valid_move(state,
        index),
5         {:ok, marker}       <- players_turn(state,
            user_id),
6         {:ok, new_state}    <- play_turn(state,
            index, marker) do
7      {:ok, new_state}
8    else
9      (other -> other)
10   end
11 end
```

When the last clause of a `with` statement is composed of a pattern identical to the predefined value to be returned by this conditional in case all checked patterns match, this clause is redundant. For example, the `with` statement within the function `update_game_state/3`, in Listing 17, has a redundant last clause, since the tuple `{:ok, new_state}` is used both as the pattern of the last clause (Line 6) and as the return value of the `with` (Line 7). This is therefore an opportunity to perform the refactoring REMOVE REDUNDANT LAST CLAUSE IN "WITH".

As shown in Listing 18, this refactoring removes the redundant last clause, leaving only two clauses in the `with` statement (Lines 4 and 5). Additionally, it also replaces the predefined return value of this conditional statement with a call to `play_turn/3` (Line 6), which was the expression previously checked in the removed redundant clause. This transformation eliminates duplicated code while maintaining the code behavior.

Listing 18 Example of a code after Remove redundant last clause in "with"

```
1  # After refactoring:
2
3  defp update_game_state(%{status: :started} =
      state, index, user_id) do
4    with {:ok, _}            <- valid_move(state,
        index),
5         {:ok, marker}       <- players_turn(state,
            user_id) do
6      play_turn(state, index, marker)
7    else
8      (other -> other)
9    end
10 end
```

Elixir provides different types of process abstractions to create concurrent systems. While `Task`[30] and `Agent`[31] are abstractions with specific purposes, `GenServer`[32] is a more generic one. Thus, GENERALISE A PROCESS ABSTRACTION is a refactoring that aims to transform `Task` or `Agent` instances into `GenServer` when these specific-purpose abstractions

30 https://hexdocs.pm/elixir/Task.html

31 https://hexdocs.pm/elixir/Agent.html

32 https://hexdocs.pm/elixir/GenServer.html

are used beyond their suggested purposes. In M26, one of the participants in the discussion that occurred in this ISSUE found in our MSR study described an opportunity to apply this refactoring:

> *"[...] If you need to monitor [a process], it is better to upgrade the* `Agent` *to a* `GenServer` *[...]"*

Listing 19 presents another opportunity for using GENERALISE A PROCESS ABSTRACTION. `DatabaseServer` is a `Task` used beyond the suggested purpose of this abstraction. According to Jurić (2024), a `Task` should be used only to execute actions without communication with other processes, since there are other process abstractions in Elixir with simpler interfaces to perform message exchange. Furthermore, `Task` is not designed to act as a long-running server, so these processes should only run a job and stop when the work is done. In Listing 19, this purpose is violated in two ways: besides the `loop/0` and `get/2` functions exchanging messages with other processes (Lines 13 and 19), the work performed by this `Task` is implemented in an infinite loop (Line 7), thus this process never stops.

The result of this refactoring is presented in Listing 20. By transforming `DatabaseServer` into a `GenServer`, which is a abstraction that remains alive waiting to serve other processes when it receives messages from them, the behavior of the `Task` was maintained. Furthermore, the message exchanges included in the actions provided by this abstraction are also preserved after the refactoring; however, they are implicit in the `GenServer.call/2` call (Line 11) and in the implementation of the `handle_call/3` callback (Line 15), thus improving the code readability. Both Listings 19 and 20 are based on an original code provided by Saša Jurić in G109 (2024).

Listing 19 Example of a code before Generalise a process abstraction

```
1  # Before refactoring:
2
3  defmodule DatabaseServer do
4    use Task
5
6    def start_link() do
7      Task.start_link(&loop/0)
8    end
9
10   defp loop() do
11     receive do
12       {:run_query, caller, query_def} ->
13         send(caller, {:query_result, run_query(
               query_def)})
14     end
15     loop()
16   end
17
18   def get(server_pid, query_def) do
19     send(server_pid, {:run_query, self(),
          query_def})
20     receive do
21       {:query_result, result} -> result
22     end
23   end
24 end
```

Listing 20 Example of a code after Generalise a process abstraction

```
# After refactoring:

defmodule DatabaseServer do
  use GenServer

  def start_link() do
    GenServer.start_link(__MODULE__, nil)
  end

  def get(server_pid, query_def) do
    GenServer.call(server_pid,{:run_query, query_def})
  end

  @impl true
  def handle_call({:run_query, query_def}, _, state) do
    {:reply, run_query(query_def), state}
  end
end
```

Since Elixir does not have an `elseif` statement, an alternative is to use nested `if..else` statements. However, this approach can result in verbose and less maintainable code, as illustrated by the `classify_bmi/2` function in Listing 21. Transform nested "if" statements into a "cond" is a refactoring found in five sources from our GLR and MSR studies. It aims to improve code readability by leveraging Elixir's `cond` statement, designed as an alternative to sequences of nested `if..else`. Listing 22 presents the refactored version of the `classify_bmi/2` function, which evaluates four distinct expressions (Lines 8 to 11) using a `cond` statement. This implementation returns the `string` corresponding to the first expression that evaluates to `true`, preserving the original behavior of the function.

Listing 21 Example of a code before Transform nested "if" statements into a "cond"

```
# Before refactoring:

def classify_bmi(weight, height) do
  {status, bmi} = calculate_bmi(weight, height)

  if status == :ok do
    if bmi < 18.5 do
      "Underweight"
    else
      if bmi < 25.0 do
        "Normal weight"
      else
        if bmi < 30.0 do
          "Overweight"
        else
          "Obesity"
        end
      end
    end
  else
    "Error in BMI calculation: #{bmi}"
  end
end
```

Listing 22 Example of a code after Transform nested "if" statements into a "cond"

```
# After refactoring:

def classify_bmi(weight, height) do
  {status, bmi} = calculate_bmi(weight, height)

  if status == :ok do
    cond do
      bmi < 18.5 -> "Underweight"
      bmi < 25.0 -> "Normal weight"
      bmi < 30.0 -> "Overweight"
      true       -> "Obesity"
    end
  else
    "Error in BMI calculation: #{bmi}"
  end
end
```

> **Finding #2:** To the best of our knowledge, we are the first to investigate specific refactorings for Elixir. Particularly, through sources from our GLR and MSR studies, we cataloged 14 refactoring strategies specific to this language (17.1% of the catalog).

3.2.3 Erlang-Specific Refactorings

This category had the lowest number of refactorings cataloged by us, 11 (out of 82). Therefore, we chose to present and describe all the Erlang- Specific refactorings in Table 9.

Despite being dynamically-typed languages, Erlang and Elixir offer `typespecs`,[33] a feature for specifying or creating the types of a function's return value and parameters. Although the compiler never uses these types specifications to optimize or modify code, they provide documentation, enhancing code readability and enabling tools such as Dialyzer[34] to perform static code analysis to detect type inconsistencies and potential bugs (Jurić 2024). Add type declarations and contracts is a refactoring found in seven sources from our SLR and MSR studies, which uses `typespecs` to create custom data types by naming recurring data structures in the codebase, as described in P22:

> *"Type declarations can give convenient names to key data structures which can then be used to document function and module interfaces. Such type information can then be used by Dialyzer to detect interface violations without occurring any runtime overhead [...]"*

Listing 23 presents the definition of `set_background/1` (Line 4), which is an opportunity to apply Add type declarations and contracts. According to the type specification on Line 3, this function receives a `tuple` composed of three `integer` values as a parameter and returns a value of the `atom` type. Although the function `set_background/1` presented in Listing 23 is already documented using a `@spec` directive (Line 3), this documentation can be improved by defining a custom data type, as shown in Listing 24. Since the `tuple` received as a parameter by `set_background/1` represents a color in the RGB standard, the directives `@typedoc` (Line 3) and `@type` (Line 6) were used to document and define a new type `color()`, respectively. This new type then replaces the explicit use of the `tuple` in the specification of `set_background/1` (Line 8).

[33] https://hexdocs.pm/elixir/typespecs.html

[34] https://www.erlang.org/doc/man/dialyzer.html

Table 9 Erlang-Specific Refactorings compatible with Elixir

Refactoring	Description	#
Typing parameters and return values	Uses Erlang/Elixir `typespecs` in a function definition to specify the types of its parameters and its return value	8
Add type declarations and contracts	Creates custom data types using Erlang/Elixir `typespecs`, thereby naming recurring data structures in the codebase	7
From meta to normal function application	Replaces calls to `Kernel.apply/3` with direct calls to `functions` that have `modules`, names, and parameter lists defined at compile time	6
Introduce processes	Introduces new concurrent processes to achieve a better mapping between parallel processes and parallel activities of the problem being solved	5
Remove processes	Removes unnecessary concurrent processes and replaces them with Elixir regular `modules`	5
Moving error- handling mechanisms to supervision trees	Removes error-handling mechanisms in a process (*i.e.,* `try..rescue`) and adds this process to a supervision tree, thereby providing the non-defensive programming style—*a.k.a. Let it crash*	3
Add a tag to messages	Identifies groups of messages exchanged between processes by adding tags. This identification allows for different treatments of received messages	1
Behaviour extraction	Extracts a `function` that repeats in different `modules` but serves specific roles in each of them. This `function` is then abstracted into an Erlang/Elixir `behaviour`, thus standardizing a contract to be followed by all `modules` that implement it	1
Behaviour inlining	In Erlang/Elixir, a `behaviour` is an interface that a module (*i.e.,* `behaviour instance`) can implement to define `functions`. This refactoring removes callbacks in a `behaviour instance`, moving the functions that implement them to the `behaviour definition`	1
Register a process	Assigns a user-defined name to a process ID and use that name instead of the process ID in messages exchanged between processes	1
Remove unnecessary calls to length/1	Replaces unnecessary calls to `length/1` in guard clauses with pattern matching	1

#: Number of sources

Listing 23 Example of a code before Add type declarations and contracts

```
\# Before refactoring:

@spec set_background({integer(), integer(),
   integer()}) :: atom()
def set_background(rgb) do
  do_something()
end
```

Listing 24 Example of a code after Add type declarations and contracts

```
\# After refactoring:

@typedoc """
    A tuple with three integer elements between 0..255
  """
@type color() :: {red :: integer(), green :: integer(), blue :: integer()}

@spec set_background(color()) :: atom()
def set_background(rgb) do
  do_something()
end
```

In Erlang and Elixir, the function `Kernel.apply/3` can be used to dynamically make a runtime decision about which function to call. This function takes as parameters the `module` name where the dynamically called function is defined, an `atom` containing the name of this function, and a `list` containing the arguments to be passed to the called function. This dynamic call pattern is known as MFA—*module*, *function*, and *arguments* (Jurić 2024). Listing 25 shows an example of using the `Kernel.apply/3` to dynamically call the function `Enum.sort/1`.

FROM META TO NORMAL FUNCTION APPLICATION is a refactoring found in six sources across all three of our studies, which allows replacing the use of `Kernel.apply/3` with a direct call to a function that has the module, name, and argument list defined at compile time, as shown in Listing 26. This refactoring improves code readability by explicitly stating which functions are being called. Therefore, it should be used whenever `Kernel.apply/3` is unnecessarily employed to perform function calls that do not depend on runtime decisions.

Listing 25 Example of a code before From meta to normal function application

```
# Before refactoring:

Kernel.apply(Enum, :sort, [[3, 4, 1, 2]])
```

Listing 26 Example of a code after From meta to normal function application

```
# After refactoring:

Enum.sort([3, 4, 1, 2])
```

Another refactoring—INTRODUCE PROCESSES—was found in five sources from our SLR and MSR studies. In P38, the authors describe the motivation for applying this refactoring as follows:

> *"[...] Introducing concurrent processes so as to achieve a better mapping between the parallel processes and the truly parallel activities [...]"*

Listing 27 shows a code where the use of INTRODUCE PROCESSES can be beneficial. The function `start/0` (Line 6) starts a process of type `GenServer` and registers it with the name __MODULE__ (*i.e.,* `Todo.Database`) (Line 7). This ensures that there is only one instance of this process throughout the system. Since this process is responsible for providing access to the system's database for all its clients, bottlenecks can occur if a large number of calls to the function `store/2` (Line 10) occur simultaneously. This can lead to an accumulation of requests in the message queue of this single process, potentially reaching a point where it cannot handle new calls of `store/2` before the previous ones finish, thus reducing the system's responsiveness.

Since the calls to `store/2` are handled by the implementation of the callback `handle_cast/2`, as shown by the tuple `{:store, key, data}` received as its first parameter (Line 15), we can refactor this callback. Listing 28 shows the resulting code of this refactoring, where a new `Task` process has been introduced in each call to `handle_cast/2` (Line 9), allowing each call to `store/2` to be handled by a different process executing concurrently, thus improving scalability. Both Listings 27 and 28 are based on an original code provided by Saša Jurić in G109 (2024).

Listing 27 Example of a code before Introduce processes

```
1  # Before refactoring:
2
3  defmodule Todo.Database do
4    use GenServer
5
6    def start do
7      GenServer.start(__MODULE__, nil, name:
         __MODULE__) #<-- Singleton!
8    end
9
10   def store(key, data) do
11     GenServer.cast(__MODULE__, {:store, key,
         data})
12   end
13
14   @impl true
15   def handle_cast({:store, key, data}, state) do
16     key
17     |> file_name()
18     |> File.write!(:erlang.term_to_binary(data))
19
20     {:noreply, state}
21   end
22 end
```

Listing 28 Example of a code after Introduce processes

```
1  # After refactoring:
2
3  defmodule Todo.Database do
4    use GenServer
5    ...
6
7    @impl true
8    def handle_cast({:store, key, data}, state) do
9      Task.start(fn ->      #<-- Introduced process
         !
10       key
11       |> file_name()
12       |> File.write!(:erlang.term_to_binary(data
           ))
13     end)
14
15     {:noreply, state}
16   end
17 end
```

> **Finding #3:** Among the 82 refactorings cataloged, 11 belong to this Erlang-specific category (13.4%), which has the fewest representatives.

3.2.4 Traditional Refactorings

Overall, we cataloged 25 Traditional refactorings compatible with Elixir. Therefore, for the same reason stated in Subsection 3.2.1, we chose to present in Table 10 only those found in at least three sources. Due to this, eight refactorings from this category were not included

Table 10 Traditional Refactorings compatible with Elixir

Refactoring	Description	#
Rename an identifier	Changes the name of an identifier such as a `function` or `module` to a more meaningful one for humans	73
Extract function	Extracts a code from a `function` and creates a new `function` with the extracted code, giving it a name that clearly explains its purpose. In the original `function`, the extracted code block is replaced by a call to the new `function`	69
Folding against a function definition	Removes duplicated code, replacing a set of expressions with a call to an existing `function` that performs the same as the duplicated code	34
Moving a definition	Moves a definition (*e.g.,* `function`, or `macro`) between `modules`	29
Add or remove a parameter	Used when it is necessary to request additional information from the callers of a `function` or when some information passed by the callers is no longer necessary	26
Inline function	This refactoring replaces all calls to a `function` with its body and removes the `function`	26
Remove dead code	Eliminates code definitions that are not being used	24
Introduce a temporary duplicate definition	Temporarily duplicates a definition when we want to test a modification. Once this new code version is approved, it will replace the original one	21
Grouping parameters in tuple	Groups a number of a function's parameters into a `tuple`	12
Extract expressions	Extracts unavoidably large and hard-to-understand expressions into smaller parts and assigns them to variables	10
Reorder parameter	Reorders parameters of a `function` that are defined in an order that doesn't group similar semantic concepts	10
Remove import attributes	Removes the import directives in a `module`, replacing all calls to imported `functions` with fully-qualified name calls (*i.e.,* `Module.function(args)`)	8
Remove nested conditional statements in function calls	Replaces nested conditional statements (*e.g.,* `case`), used as a parameter of a function call, with strict equality comparisons (*i.e.,* `===`)	6
Extract constant	Replaces occurrences of magic numbers directly in expressions with a constant (*i.e.,* `module` attribute)	5
Introduce import	Replaces fully-qualified name calls of `functions` from other modules (*i.e.,* `Module.function(args)`) with calls that use only the imported names	5
Splitting a large module	Splits a large `module` into several new ones	5
Temporary variable elimination	Removes variables solely responsible for storing results to be returned by a `function` or intermediate values	3

#: Number of sources

in these tables; however, their descriptions can be found in Table 18 in Appendix of this work.

The refactoring found in the highest number of distinct sources was RENAME AN IDENTIFIER. As shown in Table 10, it was found in 73 sources from both our SLR, GLR, and MSR studies. According to Murphy-Hill et al. (2012) and Golubev et al. (2021), this refactoring is the most frequently performed by developers. The significant number of sources extracting indicates that this fact can also be observed in our work. In Elixir, besides `modules` and `functions`, identifiers can refer to `macros`, `variables`, `map/struct fields`, `processes`, and others, as shown in the variations of this refactoring in Table 1.

EXTRACT FUNCTION emerged from 69 sources, making it the second most extracted refactoring among all in the catalog, highlighting the importance of this traditional strategy also in Elixir. In M12, the author of the ISSUE described performing the refactorings RENAME AN IDENTIFIER and EXTRACT FUNCTION in an Elixir app:

> *"During refactoring of an Umbrella app I was extracting some functions to a helper library [...] I renamed `Umbrella.Child.MyStruct` to `MyLib.MyStruct` [...]"*

Listing 29 illustrates an opportunity to perform EXTRACT FUNCTION. The function `ticket_booking/5` is responsible for booking an airline ticket for a passenger. All the main steps of the booking are done through a sequence of operations chained by pipe operators (Lines 4 to 7). After payment confirmation (Line 7), the booking process is finalized by returning a `tuple` containing reservation data that must be informed to the passenger. We can also observe that Lines 9 to 11 are responsible for presenting a report with this data. These Lines were preceded by a comment attempting to explain their purposes, highlighting that these expressions are misplaced within `ticket_booking/5`.

The refactored version is shown in Listing 30. In this code, a new function `report/1` receives a `tuple` as a parameter (Line 3) and Lines 9 to 11 from Listing 29 were moved to this new function (Lines 4 to 6). Additionally, the body of `ticket_booking/5` was updated to include a call to `report/1` (Line 14).

Listing 29 Example of a code before Extract function

```
1  # Before refactoring:
2
3  def ticket_booking(passenger, air_line, date, credit_card, seat) do
4    {company, contact, cancel_policy} = check_availability(air_line, date)
5                                        |> docs_validation(passenger)
6                                        |> select_seat(seat)
7                                        |> payment(credit_card)
8    #print booking report
9    IO.puts("Booking made at the company: #{company}")
10   IO.puts("Any doubt, contact: #{contact}")
11   IO.puts("For cancellations, see company policies: #{cancel_policy}")
12 end
```

Listing 30 Example of a code after Extract function

```
# After refactoring:

def report({company, contact, cancel_policy} =
    confirmation) do
  IO.puts("Booking made at the company: #{
      company}")
  IO.puts("Any doubt, contact: #{contact}")
  IO.puts("For cancellations, see company
      policies: #{cancel_policy}")
end

def ticket_booking(passenger, air_line, date,
    credit_card, seat) do
  check_availability(air_line, date)
  |> documents_validation(passenger)
  |> select_seat(seat)
  |> payment(credit_card)
  |> report()  # extracted function call!
end
```

Analyzing specifically the sources used to extract the TRADITIONAL refactorings, we can conclude that 165 (out of 239) analyzed sources contain refactorings of this category (69%). When comparing the results obtained from each refactoring prospecting method, we observe that 21 (out of 25) of our traditional refactorings (84%) were extracted in the SLR study, 12 were obtained in the GLR-three of which were extracted solely in this study-and 15 were found in artifacts from the MSR study.

> **Finding #4:** Traditional refactorings are also relevant in modern functional languages like Elixir, as 30.5% of our catalog belongs to this category, the second with more refactorings.

3.3 Threats to Validity

Construct Validity The main threat to construct validity concerns the format of the query strings used in our mixed methodological approach, which comprises a systematic literature review (SLR), a grey literature review (GLR), and a mining software repository (MSR) study. Since the combination of search terms is crucial for determining the quality of results, in our SLR, we followed the procedures proposed by Kitchenham and Charters (2007) to define relevant search terms, minimizing the risk of missing key papers or retrieving irrelevant articles. Although the absence of more specific search terms such as `"type inference"`, `"parametric polymorphism"`, `"tail recursion"`, `"pattern matching"`, and `"first-class lexical closures"` may potentially reduce the scope of our searches, we believe this is a minor risk. This is because, in the context of studies mentioning these specific terms, more generic terms such as `"Functional Language"` and `"Functional Paradigm"` are likely also used, allowing us to retrieve them through the query string defined in Listing 4. We also conducted a snowballing step to mitigate potential limitations of our SLR query string and ensure important papers were not overlooked. During this step, 13 new papers relevant to RQ1 were selected for in-depth analysis. Among these 13 studies, five (P380, P385, P388, P389, and P390) were indexed in the digital libraries we investigated but were not retrieved by our query string (Listing 4). The reason was the absence of the exact terms defined in our query string within their content. For instance, some of these papers included the terms `"Refactoring"` or `"Program Transformation"` but did not contain the term `"Functional Language"` or the name of one of the 19 languages listed in the query string. However, only two of these five papers (P380 and P388) actually contributed with refactoring extractions for our catalog. Thus, we argue that the query string used in the SLR proved effective in achieving its purpose of allowing us to retrieve a relevant number of papers for analysis (378 + 13 papers). For the same reason, we adhered to the guidelines proposed by Garousi et al. (2019) and Dabic et al. (2021) to define the query string used in our GLR and MSR studies, respectively, ensuring that the search terms and their synonyms align with the objectives of our research and do not undermine the results. Despite our care in selecting important keywords to compose the query strings, there is a threat that documents discussing refactorings by referring to them only by their names (*e.g.,* EXTRACT FUNCTION) may not have been retrieved by our searches. However, we believe this risk is small because in the survey with developers (Section 4), we did not receive any comments about missing refactorings.

Conclusion Validity Considering that documents retrieved in a GLR and an MSR study are not peer-reviewed, there is a risk related to the quality of their contents. To mitigate this risk, in our GLR, we established validity criteria to be applied during the document selection process. Therefore, the two authors of this work conducted an initial review of all the retrieved documents, selecting only those that are relevant and authored by professionals who are actively engaged in or have expertise with Elixir. This same risk was mitigated in the MSR study by limiting the searches to only the 10 most popular Elixir repositories on GitHub. Since these repositories have large and active communities regulating their activities, we understand that all artifacts present in them have their quality validated.

Internal Validity Threats to internal validity may be related to methodological decisions that lead to biased results. In our work, the entire process of analysis, extraction, classification, and adaptation of refactoring strategies for Elixir was manual, which can be subjective. Both authors participated in these activities to prevent these qualitative decisions from representing only one researcher's opinion, potentially biased due to their background. Thus, all steps of selection, data extraction, classification, and mapping to Elixir were independently validated by the second author, therefore minimizing individual perception biases. Hence, the decisions made in these steps reflected the consensus of both authors. Another threat to internal validity concerns possible missed refactorings in the artifacts analyzed in our MSR study. As explained in step 10 of Section 3.1, we only analyzed the titles and textual descriptions of COMMITS and PULL REQUESTS during artifact selection; however, some of these artifacts have titles and descriptions that merely indicate that a refactoring occurred without providing information on the actual code transformations performed. Thus, although there is a risk that some refactorings with potential for cataloging were missed, we claim this risk is not relevant because our data suggests that the mixed methodological approach used in this work was close to saturation. While our MSR study had the highest number of documents retrieved in searches (488 artifacts), it also contributed the least to adding new refactorings to the catalog (six out of 82). Furthermore, another mitigating factor for this risk is that we conducted a survey with 151 developers and also published our refactorings in a public GitHub repository (Vegi and Valente 2023a), which was widely consulted. Even so, we did not receive any suggestions for new refactorings beyond the 82 cataloged. Although this does not fully guarantee the completeness of our list of refactorings, it demonstrates that the risk of omitting important refactorings in Elixir is low.

External Validity The threat to external validity concerns the generalizability of our findings. Considering that there are other digital libraries of scientific papers beyond the four utilized in our SLR, there is a risk that important works may not have been retrieved in our research. To mitigate this threat, we incorporated a snowballing step into our methods, enabling the retrieval of relevant papers published in a broader range of digital libraries. In our GLR, we used only Google as the web search engine, which carries the risk of overlooking important grey literature documents indexed only in other search engines. However, we believe this risk is minimal, as Google currently indexes approximately 50 billion documents to search through, the largest number among search engines. In comparison, Bing indexes about 1.2 billion documents, making it the second-largest source of indexed content.[35] Finally, in our MSR study, we have a similar concern about generalizability. We selected only 10 repositories to investigate refactorings for Elixir, a small sample considering there are approximately 93K Elixir repositories on GitHub. Therefore, our dataset may not fully represent the entire population of Elixir-based projects. However, it is important to emphasize that the refactorings were also extracted from other sources (SLR and GLR) and then validated through a survey with developers (Section 4). In this survey, we did not receive any comments about missing refactorings.

4 Catalog Validation

In this section, we introduce the second study of our work, in which we surveyed Elixir developers to validate the catalog of refactorings outlined in Section 3. Specifically, we gathered quantitative data using this instrument to address the following research question:

***RQ2.* What are the developers' perceptions of refactorings in Elixir?** In this RQ, we aim to understand, from the developers' perspective, the prevalence and relevance of each refactoring in real-world Elixir projects. The way we define relevance, as explained to survey participants, pertains to how a refactoring strategy can potentially improve the maintainability, comprehensibility, and evolution of Elixir code. The prevalence refers to the frequency with which each refactoring is performed in Elixir codebases.

Since this validation entails the active involvement of human subjects, prior to conducting the survey, it underwent evaluation and approval by the Research Ethics Committee at the Federal University of Minas Gerais (Brazil).

We dedicate Section 4.1 to outline the methodology employed in designing, conducting, and analyzing our survey results. In Section 4.2, we detail and discuss upon the findings regarding the prevalence and relevance of refactorings in Elixir. Finally, Section 4.3 addresses potential threats to the validity of our study and what measures we took to mitigate them.

4.1 Survey Design

While our process of cataloging refactorings for Elixir also involved analyzing documents and artifacts created by developers working with this language, we did not have direct contact with these developers during this process. This limitation prevented us from gaining a broader understanding of the motivations behind each code transformation and the extent of

[35] https://www.worldwidewebsize.com/

the impacts caused by these changes on code quality. Additionally, the refactorings identified in our SLR were all adaptations of code transformations performed in other functional languages, so we did not have confirmation of their impacts on real-world Elixir projects. Figure 3 summarizes the steps we followed to conduct a survey that allowed us to have contact with developers to elucidate these issues. We also elaborate on these steps in the following paragraphs.

1) Survey Structure Our survey questionnaire commenced by outlining the study's objectives, introducing the research team, explaining the voluntary nature of participation, and obtaining participants' consent. The questions posed to participants during this study are available in our replication package (Vegi and Valente 2024). Specifically, these questions were grouped into the following topics:

- *Demographics*: First, we inquired about the participant's geographical location, their years of experience with Elixir, and the number of Elixir projects they have been involved in.
- *Perceptions on refactorings for Elixir*: Afterward, we introduced a compilation of refactorings tailored for Elixir. This compilation comprised explanations of these transformations and illustrative code snippets depicting the code before and after each refactoring. We then inquired about the frequency with which participants encounter each refactoring in their Elixir projects and the positive impacts they believe these refactorings have on these projects. Participants responded using a scale ranging from one (*very low*) to five (*very high*). To prevent any bias from the presentation order of the refactorings, they were displayed in random order. Furthermore, to maintain the integrity of our findings, questions regarding developers' perceptions of refactorings were optional, granting participants the freedom to skip questions related to transformations they were unfamiliar with.
- *Final remarks*: At the end of the questionnaire, participants were given the option to leave comments, justifying their responses.

Given that it would be time-consuming and tiring for participants to provide their perceptions on each of the 82 refactorings cataloged for Elixir, we chose to create five different versions of the survey, with each version asking participants about their perceptions of a maximum of 17 refactorings. Therefore, each refactoring was included in just one survey version, and the choice of refactorings included in each version was randomized. The distribution of refactorings across the survey versions is shown in Table 11.

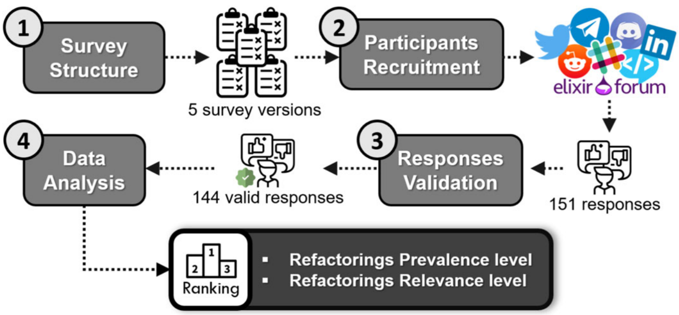


Fig. 3 Overview of survey on refactorings for Elixir

Table 11 Survey versions template

Version	# Refactorings by type
A	(5 Traditional) + (6 Functional) + (2 Erlang-Specific) + (3 Elixir-Specific)
B	(5 Traditional) + (6 Functional) + (2 Erlang-Specific) + (3 Elixir-Specific)
C	(5 Traditional) + (6 Functional) + (2 Erlang-Specific) + (3 Elixir-Specific)
D	(5 Traditional) + (7 Functional) + (2 Erlang-Specific) + (3 Elixir-Specific)
E	(5 Traditional) + (7 Functional) + (3 Erlang-Specific) + (2 Elixir-Specific)

A distinct Google form was utilized to create each survey version. Moreover, we developed a simple script—available in our replication package (Vegi and Valente 2024)—that yields a random number between zero and four. Based on this number, the script also forwards the participants to the survey version they should answer to. This survey structure is similar to one used in our previous study on code smells for Elixir (Vegi and Valente 2023).

2) Participants Recruitment To gather findings that represent the perceptions of a diverse group of developers, we prioritized recruiting a substantial number of Elixir developers with different levels of experience in this language, diverse cultural backgrounds, and varying number of Elixir projects they were working on.

Our recruitment process followed the same approach used in our previous study on code smells for Elixir (Vegi and Valente 2023), which was based on promotion across social media platforms. Initially, we posted an invitation on Twitter/X for Elixir developers to participate in our survey and encouraged them to share it within their networks. We utilized in this post popular hashtags from the Elixir developer community (*i.e., #MyElixirStatus* and *#ElixirLang*) to maximize visibility. Concurrently, we promoted the survey on the Elixir Forum,[36] the official discussion platform for Elixir developers. Additionally, throughout the two-week survey period, we regularly posted invitations on other communication channels of the language, including DevTalk,[37] Discord,[38] LinkedIn,[39] Reddit,[40] Slack,[41] and Telegram.[42]

We concluded the survey with 151 responses after two weeks. The recruited participants are spread across 42 countries spanning all continents. As depicted in Fig. 4, most of our respondents are located in America (46.5%) and Europe (41%), with the United States (22.9%) and Canada (11.1%) standing as the top two countries.

Regarding their familiarity with Elixir, about 78% of our participants have worked with Elixir for over three years, and 66% have used this language on more than four distinct projects.

3) Responses Validation To eliminate potentially biased responses, we cross-referenced the names of the authors of the 239 sources selected for cataloging refactorings in our SLR, GLR, and MSR studies with all survey respondents' identities. After conducting this analysis,

36 https://elixirforum.com/
37 https://devtalk.com/elixir
38 https://discord.gg/elixir
39 https://www.linkedin.com/groups/6530248/
40 https://www.reddit.com/r/elixir/
41 https://elixir-slackin.herokuapp.com
42 https://t.me/elixir_world

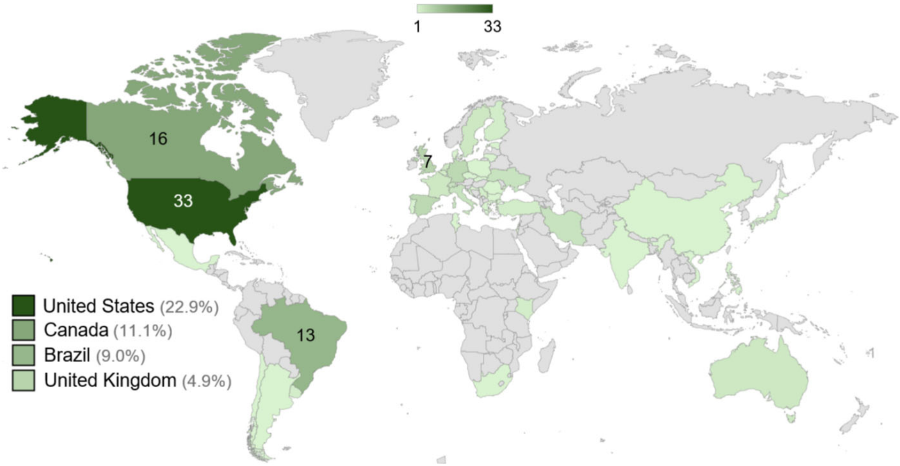


Fig. 4 Countries where the survey participants reside. The legend presents the top 4 countries with most participants

we discovered that seven (out of 151) participants are also authors of sources used in the prospecting of refactorings. Four of them are the authors of documents selected in the GLR, and three are authors of artifacts from our MSR study. Therefore, we filtered out these seven responses, keeping 144 valid responses for further analysis.

4) Data Analysis Given that we divided the 82 refactorings into five survey versions and assigned a version randomly to each participant (step 1), the number of responses received for each version, and consequently for each refactoring, varied. This variation is shown in Table 12. To compensate for this variation, we calculated the arithmetic mean of all valid responses, thereby normalizing the prevalence and relevance levels of each one of these code transformations. As a result, the closer a refactoring's prevalence or relevance level is to five, the more common or beneficial it can be considered for Elixir systems, respectively.

Despite the fact that the questions about the relevance and prevalence of each refactoring were optional, the average percentage of valid (*i.e.,* non-empty) answers for all refactorings analyzed in our survey was 97.6%, with a standard deviation of 2.5%. Among all those cataloged, the INLINE MACRO refactoring had the lowest response rate (90.9%). On the other hand, 34 out of 82 refactorings (41.5%) had a 100% response rate.

Finally, we employed the Mann-Whitney test (Siegel and Castellan 1988) to evaluate whether the respondents' experience level affected their perception of the prevalence and relevance of refactorings. Due to the lack of a normal distribution in our data, we opted for this non-parametric test. The number of years a participant has worked with Elixir (*at most three* or *more than three*) and the number of projects they have worked on using this language (*at most four* or *more than four*) were the two distinct grouping variables used in this test.

Table 12 Valid responses for each survey version

Version A	Version B	Version C	Version D	Version E
23	32	33	30	26

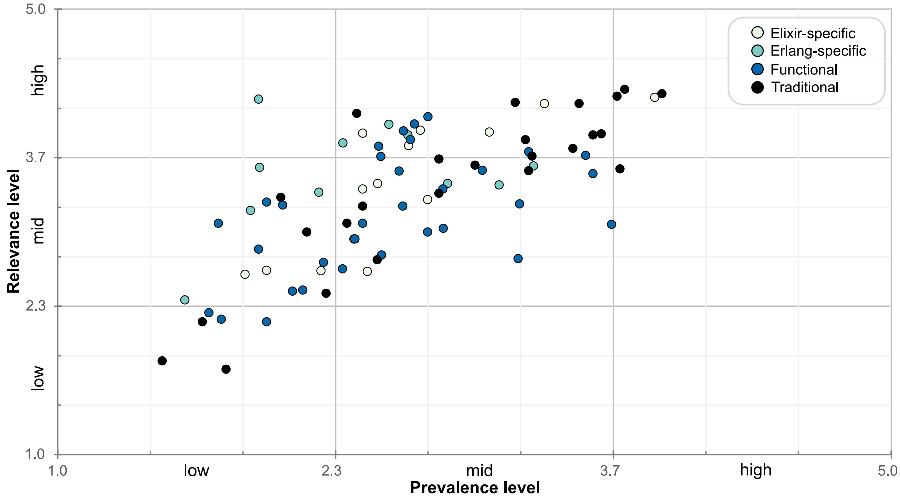


Fig. 5 Developers' perception of refactorings in Elixir (RQ2)

We conducted these analyses using the SPSS statistical analysis tool,[43] with a significance level set at 0.05.

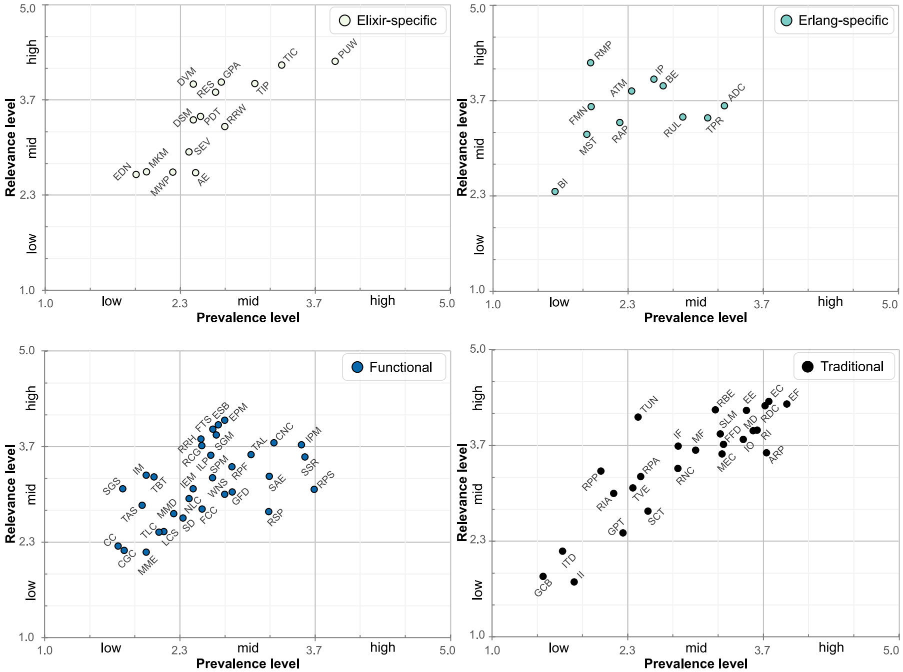
4.2 What are the developers' perceptions of refactorings in Elixir? (RQ2)

To facilitate the comparison between the refactorings for Elixir, we chose to present the results of our survey using scatter plots. Due to the large number of refactorings in our catalog, two different perspectives were used for this presentation. In Fig. 5, we show a scatter plot containing all 82 refactorings from the catalog, thus providing an overview of the results. On the other hand, Fig. 6 presents a scatter plot for each of the four refactoring categories, thus improving the readability of our results. To further illustrate the relevance and prevalence levels perceived for each refactoring, we also split the planes of these plots into nine zones. Specifically, there are three distinct quadrants for each axis: LOW (average scores between 1 and 2.3), MID (average scores between 2.3 and 3.7), and HIGH (average scores between 3.7 and 5). Since the survey responses ranged from 1 to 5, the averages presented in the scatter plots also fall within this range. Consequently, each quadrant has a length of approximately 1.33, resulting from an equal division of the range.

When analyzing each of the nine zones in Fig. 5 separately, we can observe that the one with the highest concentration is the central one (MID-prevalence and MID-relevance), resulting in a cluster composed of 29 (out of 82) refactorings. As can be better observed in Fig. 6, the FUNCTIONAL refactorings are the most abundant in the three MID-prevalence level zones (22 out of 53). Additionally, this is also the category with the most representatives in the central zone of the scatter plot, as 48.3% (14 out of 29) of the refactorings in this zone are functional.

Figure 5 also shows that 70.6% (58 out of 82) of the refactorings in our catalog have at least MID-prevalence, with five of them—four traditional and one Elixir-specific—having HIGH-prevalence. Moreover, Fig. 6 shows that the three most prevalent refactorings are the

43 https://www.ibm.com/spss



ADC : Add type declarations and contracts
ARP : Add or remove a parameter
BE : Behaviour extraction
CC : Closure conversion
CNC : Convert nested conditionals to pipeline
DVM : Default value for an absent key in a map
EDN : Explicit a double boolean negation
EF : Extract function
ESB : Eliminate single branch
FFD : Folding against a function definition
FTS : From tuple to struct
GFD : Generalise a function definition
GPT : Grouping parameters in tuple
IF : Inline function
ILP : Improving list appending performance
IO : Introduce overloading
IPM : Introduce pattern matching over a parameter
LCS : List comprehension simplifications
MEC : Move expression out of case
MKM: Modifying keys in a map
MME: Merging match expressions into a list pattern
MWP: Moving "with" clauses without pattern matching
PDT : Pipeline for database transactions
RAP : Register a process
RCG : Replace nested conditional in "case" with guards
RES : Replace "Enum" collections with "Stream"
RIA : Remove import attributes
RNC : Remove nested conditionals in function calls
RPF : Replace pipeline with a function
RPS : Replace a call with raw value in a pipeline start
RRH : Replace recursion with a higher-level construct
RUL : Remove unnecessary calls to length/1
SCT : Simplifying checks by using truthness condition
SEV : Simplifying Ecto schema fields validation
SGS : Simplifying guard sequences
SPM : Simplifying pattern matching with nested structs
TAL : Turning anonymous into local functions
TBT : Transform a body-recursive to a tail-recursive
TIP : Transform "if" using pattern matching into "case"
TPR : Typing parameters and return values
TVE : Temporary variable elimination
AE : Alias expansion
ATM : Add a tag to messages
BI : Behaviour inlining
CGC : Converts guards to conditionals
DSM : Defining a subset of a map
EC : Extract constant
EE : Extract expressions
EPM : Equality guard to pattern matching
FCC : Function clauses to/from case clauses
FMN : From meta to normal function
GCB : Group case branches
GPA : Generalise a process abstraction
IEM : Introduce Enum.map/2
II : Introduce import
IM : Inline macro
IP : Introduce processes
ITD : Introduce a temporary duplicate def.
MD : Moving a definition
MF : Move file
MMD: Merging multiple definitions
MST : Move error-handling to supervision tree
NLC : Nested list functions to comprehension
PUW : Pipeline using "with"
RBE : Reducing a boolean equality expression
RDC : Remove dead code
RI : Rename an identifier
RMP : Remove processes
RPA : Reorder parameter
RPP : Replace conditional with polymorphism
RRW : Remove redundant last clause in "with"
RSP : Remove single pipe
SAE : Struct field access elimination
SD : Splitting a definition
SGM : Struct guard to matching
SLM : Splitting a large module
SSR : Static structure reuse
TAS : Transforming list appends and subtracts
TIC : Transform a nested "if" into a "cond"
TLC : Transform to list comprehension
TUN : Transform negated "unless" into a "if"
WNS : Widen or narrow definition scope

Fig. 6 Developers' perception of refactorings in Elixir by category (RQ2) traditional ones, including Extract function (EF, 3.90)[44] and Extract constant (EC, 3.72), along with the Elixir-specific refactoring Pipeline using "with" (PUW, 3.86). On the other hand, the three least prevalent ones are the traditional refactorings Group case branches (GCB, 1.50) and Introduce a temporary duplicate definition (ITD, 1.69), along with the Erlang-specific Behaviour inlining (BI, 1.61). Overall, considering that the use of these code transformations is not uncommon in systems developed in Elixir, it is important for developers working with this language to master these techniques, thereby acquiring the capacity to refactor their own code and to conduct code reviews more productively, as a quick understanding of the code changes made by their teammates can save work time.

> **Finding #5:** Most of the refactorings in our catalog have prevalences between MID and HIGH (70.6%), so it is important for developers to master these techniques because they are commonly used in Elixir projects.

When comparing refactoring categories, our data suggests that traditional refactorings are the most frequently used in Elixir, since their average prevalence level is 2.88, the highest value among all categories. In contrast, Erlang-specific refactorings had the lowest average prevalence level (2.42). This is also evident when observing the distribution of refactorings in the respective plots for each category (see Fig. 6). While 16% (4 out of 25) of traditional refactorings have HIGH-prevalence, the Erlang-specific category has the highest percentage of refactorings with LOW-prevalence level, at 45.5% (5 out of 11).

Regarding the relevance of refactorings, the majority of these transformations (92.7%) have at least MID-relevance. Additionally, in Fig. 5, we can see that 29 of them (35.3%) are in HIGH-relevance zones, showing that, in the perception of developers who participated in the survey, a significant portion of our catalog has a non-negligible potential to improve the quality of systems developed in Elixir. Specifically, participants believe that the three refactorings with the highest levels of relevance are respectively Extract constant (EC, 4.28), Extract function (EF, 4.24), and Remove dead code (RDC, 4.22), all of which are traditional refactorings. The three least relevant ones are also traditional refactorings, Introduce import (II, 1.77), Group case branches (GCB, 1.84), and Introduce a temporary duplicate definition (ITD, 2.19). Although Erlang-specific refactorings had the lowest average prevalence level, our data suggests that this category has the highest average relevance level (3.53), making it the one with the greatest potential to improve the quality of systems developed in Elixir. On the other hand, functional refactorings were those with the lowest average relevance level according to the participants' perception (3.16).

> **Finding #6:** Most of the refactorings in our catalog have relevances between MID and HIGH (92.7%), therefore having a non-negligible potential to improve the quality of Elixir systems.

To better understand which refactorings in our catalog require more attention from developers, we calculated the arithmetic mean of their relevance and prevalence levels. In other words, our goal in calculating this mean for each refactoring was to highlight which refactorings in the catalog stand out the most overall, not just in one of the two characteristics

[44] When discussing each refactoring, we are adding between parentheses the acronym used in Fig. 6 (*e.g.,* EF) and the respective average score of the survey answers (*e.g.,* 3.90).

Table 13 Refactorings with a high mean between their relevance and prevalence levels

Refactoring (Acronym)	Category	Mean
Extract function (EF)	TR	4.07
Pipeline using "with" (PUW)	ES	4.03
Extract constant (EC)	TR	4.00
Remove dead code (RDC)	TR	3.95
Extract expressions (EE)	TR	3.83
Transform nested "if" statements into a "cond" (TIC)	ES	3.74
Rename an identifier (RI)	TR	3.74
Moving a definition (MD)	TR	3.72
Reducing a boolean equality expression (RBE)	TR	3.68

TR: Traditional. / ES: Elixir-Specific

evaluated in our survey (prevalence or relevance). As shown in Table 13, nine (out of 82) refactorings have HIGH-means (*i.e.,* scores above 3.67)—seven traditional and two Elixir-specific—and, therefore, are those that deserve special attention from Elixir developers. The refactoring EXTRACT FUNCTION, for example, has a prevalence level of 3.90 and a relevance level of 4.24, as shown in Fig. 5. For this reason, this code transformation strategy obtained a mean of 4.07 (*i.e.,* $(3.90 + 4.24)/2$), the highest among all the refactorings in our catalog.

Lastly, to understand which categories stood out the most, we calculated the arithmetic mean of each category by averaging its relevance and prevalence levels. As shown in Table 14, the category with the highest arithmetic mean was TRADITIONAL refactorings (3.15). As detailed in our replication package, the mean relevance of all refactorings in this category is 3.42, and its mean prevalence level is 2.88. Therefore, this category obtained a mean of 3.15 (*i.e.,* $(3.42 + 2.88)/2$). It was closely followed by the ELIXIR- SPECIFIC category (3.01), the second-highest mean. Not coincidentally, these are also the categories of the nine refactorings with HIGH-means.

> **Finding #7:** Nine refactorings in the catalog (11%) have a HIGH arithmetic mean between their relevance and prevalence levels, making them the ones that require the most attention from Elixir developers, both to learn and to perform.

In the Mann-Whitney test, the mean rank helps determine which of the two groups being compared tends to have higher or lower values relative to the other (Siegel and Castellan 1988). Our results indicate that the developers' experience level affected how they perceived just 16 refactorings from our catalog. Table 15 shows how their perception of 11 out of these 16 refactorings is influenced by the number of years they had worked with Elixir. Out of these

Table 14 Means between the relevance and prevalence levels of each category

Category of refactorings	Mean
TRADITIONAL	3.15
ELIXIR- SPECIFIC	3.01
ERLANG- SPECIFIC	2.97
FUNCTIONAL	2.88

Table 15 Influence of the Elixir experience in the developer's perception of a refactoring

Refactoring (Perception)	Mean rank ≤ 3 years	 > 3 years	*Sig.*
From tuple to struct (*P*)	8.71	***18.68***	0.011
Introduce a temporary duplicate definition (*P*)	9.00	***15.50***	0.047
Transform a body-recursive function to a tail-recursive (*P*)	8.75	***15.61***	0.035
Behaviour inlining (*R*)	***18.90***	10.08	0.007
Closure conversion (*P*)	***21.83***	13.22	0.026
Grouping parameters in tuple (*P*)	***22.33***	12.36	0.007
Grouping parameters in tuple (*R*)	***21.33***	13.35	0.041
Modifying keys in a Map (*P*)	***25.38***	15.23	0.040
Nested list functions to comprehension (*R*)	***25.62***	15.20	0.034
Replace conditional with polymorphism via Protocols (*P*)	***22.75***	12.98	0.009
Temporary variable elimination (*R*)	***18.00***	11.50	0.047
Transform "if" statements using pattern matching into a "case" (*P*)	***22.00***	13.17	0.022

R: Relevance. / *P*: Prevalence. / *Sig.*: Significance

11 refactorings, three—FROM TUPLE TO STRUCT, INTRODUCE A TEMPORARY DUPLICATE DEFINITION, and TRANSFORM A BODY- RECURSIVE FUNCTION TO A TAIL- RECURSIVE—are viewed as more prevalent among developers who have worked with Elixir for more than three years. Conversely, those with fewer years of experience thought the other eight refactorings shown in Table 15 to be more prevalent or relevant than their more experienced counterparts.

The number of Elixir projects worked on has less influence on developers' perception of our refactorings. Table 16 shows that only five refactorings were affected by this factor. Specifically, developers with more than four Elixir projects found INTRODUCE IMPORT and INTRODUCE PROCESSES more relevant and prevalent, respectively. In contrast, those with a maximum of four Elixir projects perceived MOVE FILE, REMOVE DEAD CODE, and REPLACE "ENUM" COLLECTIONS WITH "STREAM" as more prevalent or relevant than their counterparts with more Elixir project experience.

Interestingly, experience levels with Elixir have more influence on developers' perception of the prevalence than of the relevance of these code transformations. Out of the 18

Table 16 Influence of the number of Elixir projects in the developer's perception of a refactoring

Refactoring (Perception)	Mean rank ≤ 4 projects	 > 4 projects	*Sig.*
Introduce import (*R*)	10.28	***17.74***	0.032
Introduce processes (*P*)	8.29	***17.14***	0.015
Move file (*R*)	***18.11***	10.12	0.008
Move file (*P*)	***17.22***	10.62	0.032
Remove dead code (*R*)	***14.58***	8.65	0.036
Replace "Enum" collections with "Stream" (*P*)	***14.46***	7.95	0.017

R: Relevance. / *P*: Prevalence. / *Sig.*: Significance.

perceptions[45] influenced by the two factors analyzed by the Mann-Whitney test, 11 of them (61%) are about the prevalence of refactorings in our catalog. Another aspect to highlight is that perceptions regarding TRADITIONAL refactorings were the most affected by developers' experience levels, as seven (out of 16) refactorings whose perceptions were influenced by these factors belong to this category. On the other hand, the ERLANG- SPECIFIC category had only two perceptions of refactorings influenced by experience levels with Elixir, thus being the least affected by these factors.

> **Finding #8:** The developers' experience levels influenced their perception of only 16 refactorings in our catalog (19.5%). Among these, the refactorings belonging to the traditional category and the perceptions of the refactorings' prevalence were the most affected by these factors.

The replication package[46] of this work (Vegi and Valente 2024) contains the full results of the Mann-Whitney test (including those lacking statistical significance) and the prevalence and relevance levels of each of the 82 refactorings that compose our catalog.

4.3 Threats to Validity

Construct Validity The risk of recruiting some unrepresentative participants to respond the survey poses a threat to construct validity of this study. In other words, if the recruitment process of our survey were not effective enough to primarily find experienced software developers with a deep understanding of Elixir, the findings obtained with this research tool could be inconsistent with the real world. To mitigate this threat, we followed the same strategy used in our previous study on code smells for Elixir (Vegi and Valente 2023). So, we promoted our survey mainly using Elixir's official communication channels, thereby increasing the chances of attracting respondents who fit the target profile of our research. Additionally, we examined the level of experience among participants and the diversity of projects they have worked using Elixir. A significant portion of respondents (78%) have more than three years of experience with Elixir, and 66% have engaged in over four distinct projects using this functional language.

Conclusion Validity The primary concern regarding this type of threat in our survey is the disparity in the quantity of answers obtained by each version of the questionnaire. Certain versions received more responses than others since each participant was required to respond to just one of them, and this allocation was done randomly. Nonetheless, the version of the questionnaire with fewer valid responses recruited 23 participants (16% of 144), and the version with the most responses obtained 33 (23% of the total), which are not significantly inbalanced percentages, considering that the ideal scenario would be each version getting 20% of the total survey responses. Moreover, the coefficient of variation (CV) for the number of responses among the versions was 14.6%, suggesting only moderate dispersion relative to the mean (Walpole et al. 2007). To mitigate this threat, we normalized the answers for each of our five questionnaire versions, thereby compensating for the imbalance. Another threat

[45] As can be seen in Tables 15 and 16, the refactorings GROUPING PARAMETERS IN TUPLE and MOVE FILE had both their perceptions of relevance and prevalence affected by the analyzed factors, resulting in 18 affected perceptions regarding 16 different refactorings.

[46] https://doi.org/10.5281/zenodo.11372758

to our survey's conclusion validity relates to how we recruited participants. By employing a public social media strategy, we potentially acquired biased responses from participants who are also the authors of the documents we utilized as sources for cataloging refactorings. To address this threat, we identified and excluded the seven responses received from the authors of these documents. On the other hand, we acknowledge that asking developers other than the authors of the documents about their perceptions introduces certain threats, such as a lack of authorship (since these developers might not be experts in the surveyed refactorings) and a lack of incentive (since their understanding of the surveyed refactorings is not guaranteed). However, in future surveys and studies, we also plan to consult the authors of the refactorings to mitigate these threats. This approach will also eliminate the need to divide the refactorings among respondent groups, as authors are typically proponents of a limited number of refactorings.

Internal Validity The main threat to the internal validity of this study concerns possible misunderstandings by respondents, which could influence the quality of their answers and consequently our findings. More specifically, there is a risk that participants' perception of refactorings may become distorted if they misunderstand the motivations and mechanics of these transformations. To mitigate this risk in our questionnaires, we included explanations about the refactorings and illustrative code snippets depicting the code before and after each transformation. Additionally, we made all questions about refactorings optional, giving participants the freedom to only respond about the transformations they feel confident about. Moreover, as classifying the prevalence and relevance of refactorings on a scale can be subjective, we described in the form the key points to be considered by respondents when rating each of these two characteristics. Finally, the first author of this work was available via email to clarify any doubts respondents had regarding refactorings and the questions to be answered. This set of strategies used to mitigate the aforementioned risk also partially mitigated another threat. We used the Mann-Whitney test (Siegel and Castellan 1988) to measure whether the number of years working with Elixir or the number of projects developed using this language influenced respondents' perceptions regarding the prevalence and relevance of the refactorings. Although we rigorously followed all the necessary steps for conducting this statistical test, there is a risk that other variables not considered in this analysis—such as the degree of understanding of the refactorings' documentation by the respondents—may have influenced their answers, potentially impacting our conclusions. However, it is first important to mention that the participants had the option to not answer any question in case of doubts. Moreover, no comments related to difficulties in understanding the refactorings were reported by the 151 respondents in the final remarks field of the questionnaires. Considering that we have a catalog with 82 refactorings, another threat that could affect our results is participant weariness throughout a lengthy questionnaire, as noted by Nardone et al. (2023) in other studies. To mitigate this threat, we created five versions of the questionnaire and distributed the 82 refactorings among them, thereby reducing the number of questions asked to each participant. Additionally, we randomly list the refactorings presented to each participant, thus reducing the risk that participant fatigue would consistently affect the quality of responses provided for the same refactoring strategy.

External Validity The threats to external validity concern the generalization of our results. Although we recruited 144 Elixir developers to answer our questions, this group may not fully represent the overall perception of the entire community of developers working with this language. To mitigate this threat, we promoted our survey across eight different communication channels within the Elixir community. This allowed us to recruit a representative group, consisting of developers from 42 countries spanning all continents, with distinct levels of expertise in Elixir. Additionally, it is important to emphasize that we employed the Mann-Whitney test (Siegel and Castellan 1988) to evaluate whether the respondents' experience level affected their perception of the prevalence and relevance of refactorings. The results of this test showed that only the perceptions about 16 out of 82 refactorings (19%) were influenced by these factors, thus indicating that this study is not biased by a specific group of developers.

5 Implications

Based on our results, we shed light on the following practical implications:

1. Priority in learning and understanding the refactorings Our findings can assist developers working with Elixir in deciding which refactoring techniques to learn first while studying our catalog. A useful metric to consider in making this choice is the prevalence level of the refactorings, as higher prevalence indicates more frequent use of a particular transformation in the Elixir systems maintenance. Since these frequent refactoring strategies can be executed by other team members, developers must first become proficient in them to better comprehend the code written by their teammates and thus be more efficient during a code review process, for example.

2. Priority in choosing which refactoring to perform While developers are working on maintaining and evolving an Elixir system, they may simultaneously encounter several refactoring opportunities in the code and then face the dilemma of choosing which transformation to perform first. Similar to the implication presented earlier, we conjecture that the relevance level is a good indicator to define the priority in choosing which refactoring to perform first. Since the more relevant a refactoring, the greater its potential to improve code quality, it is recommended to perform higher relevant refactorings before the others.

3. Assists beginners in Elixir to produce idiomatic code Idiomatic code refers to a coding style that follows the conventions and standards of a specific programming language. Therefore, they are more natural and efficient for that particular language, as they adhere to the best practices accepted by the developer community working with it (Zhang et al. 2022). When developers with a background in object-oriented languages (*e.g.,* Java and C++) start programming in Elixir, it is natural to expect them to try to reproduce in this language the coding styles they are accustomed to, thus producing non-idiomatic code, which can, for example, hinder collaboration among their teammates. Our catalog can therefore guide developers who are beginners in Elixir to make their code more idiomatic. Some practical examples of this implication include replacing the excessive use of classical conditional constructs (*e.g.,* `if` and `unless`) with pattern matching (see PIPELINE USING "WITH" and INTRODUCE PATTERN MATCHING OVER A PARAMETER) and replacing concurrency units with other process abstractions more appropriate for the code (see GENERALISE A PROCESS ABSTRACTION).

4. Directing efforts to adapt tools for automatically performing refactorings The STYLER[47] is an Elixir formatter plugin that can also perform seven (out of the 82) refactorings from our catalog. Although not exclusively focused on refactoring, the STYLER is

[47] https://github.com/adobe/elixir-styler

currently the main tool for this purpose among developers working with Elixir. Considering that even the primary refactoring tool for Elixir still has limited capabilities, we conjecture that our catalog of refactorings can be used to fill these gaps, improving the capacity of tools like STYLER to automatically perform a larger number of refactorings. Additionally, metrics such as the prevalence and relevance levels of refactorings can direct the efforts of the developers of these tools when making these adaptations, since the implementation of the most prevalent and relevant refactoring strategies should be prioritized.

5. Researchers should establish a relationship between refactorings and code smells for Elixir One of the main motivations for developers to perform refactorings is to improve the quality of their code, which can be done, for example, by removing code smells (Abid et al. 2020). Considering that there are now specific catalogs of code smells (Vegi and Valente 2023) and refactorings for Elixir, further studies establishing a relationship between these catalogs can be conducted. They should focus on reproducing, in the context of Elixir, the same approach used by Fowler and Beck (1999) with their well-known catalogs, where for each traditional code smell, the authors mapped refactorings to remove them.

6 Related Work

To the best of our knowledge, our study is the first one that catalogs refactorings for Elixir. Other research, however, provides transformation strategies and tools for other functional languages like Erlang (Avgerinos and Sagonas 2009; Bozó and Tóth 2009; Drienyovszky et al. 2010; Héder 2007; Király 2013; Kozsik et al. 2007; Li and Thompson 2012; Li et al. 2014, 2006, 2008; Lövei et al. 2008, 2007b, a; Sagonas and Avgerinos 2009; Sagonas and Luna 2008; Taylor and Derrick 2015; Thompson et al. 2016), Haskell (Brown 2008; Brown et al. 2011; Erwig and Ren 2007; Farmer et al. 2012, 2015; Gill 2006; Guttmann et al. 2003; Lämmel 2002; Lee 2011; Li et al. 2005; Sculthorpe et al. 2013; Thompson 2005; Thompson and Reinke 2003; Tullsen 2022; Villavicencio 2011), ML (Bellegarde 1995; Johnsson 1985), OCaml (Rowe et al. 2019a, b), Scala (Sherwany et al. 2015), and Racket (Chang 2013). In addition, some studies have investigated refactorings in Ruby, which is a non-functional language that influenced the creation of Elixir's syntax (Arnaoudova and Constantinides 2008; Corbat et al. 2007).

Li and Thompson (2012); Li et al. (2006, 2008) introduced an automatic refactoring tool for Erlang called WRANGLER, demonstrating its use through some refactoring strategies for this functional language. Similarly, Sagonas and Avgerinos (2009) and Lövei et al. (2008) proposed the tools TIDIER and REFACTORERL, respectively, along with code transformations for Erlang. Concerning research on Haskell refactorings, Brown (2008); Brown et al. (2011) and Li et al. (2005) briefly described the refactorings implemented in the HARE tool, a Haskell refactorer. Li and Thompson (2006) compared the refactoring strategies for Haskell and Erlang discussed in the previously mentioned studies, highlighting that each language has its unique constraints and challenges, thus justifying the creation of language-specific refactoring catalogs.

Some studies delve into even more specific refactoring strategies for functional languages. For instance, there are refactorings designed to parallelize programs in both Erlang (Bozó et al. 2015; Bozó et al. 2014; Brown et al. 2014, 2011; Janjic et al. 2021; Kozsik et al. 2018, 2017; Tóth et al. 2017) and Haskell (Barwell et al. 2018; Brown et al. 2011). Additionally, Chechina et al. (2017) presented a systematic and tool-based approach for refactoring distributed Erlang applications. This approach uses some traditional refactorings that also belong to our catalog, such as Rename an Identifier, Extract Function, and Moving a Definition. Furthermore, Li and Thompson (2015) introduced three refactorings for promoting concurrency in Erlang applications. All of these refactorings are compatible with Elixir and have therefore been incorporated into our catalog.

The aforementioned studies were retrieved by our systematic literature review (Section 3.1), and some refactoring strategies outlined in them were adapted to our catalog of refactorings for Elixir, as presented in the replication package of this work (Vegi and Valente 2024). Other studies have also conducted SLRs or GLRs to explore different areas related to refactoring. Aiming to understand the field and existing research results, Abid et al. (2020) reviewed 3,183 papers on refactoring published until May 2020. Based on this SLR, the authors created a taxonomy to classify the existing research in this area, identified research trends, and highlighted gaps in the literature to be filled in further research. In another study, Al Dallal and Abdin (2018) conducted an SLR focused on exploring works related to the impacts of refactorings on internal and external software quality attributes. The authors identified 76 papers published until December 2015 in seven relevant digital libraries. Through an in-depth analysis of these papers, they provided a clear view of the impacts of some refactoring strategies on quality attributes such as cohesion, coupling, complexity, and size. Similarly, Singh and Kaur (2018) conducted an SLR on the relationship between refactorings and code smells in object-oriented systems. After applying their inclusion-exclusion criteria, the authors selected and analyzed 238 papers published until September 2015 in four digital libraries. Among other findings, this study lists a series of refactoring tools and describes which code smells each of them can remove. Finally, Abid et al. (2022) performed a GLR on over 100K Stack Overflow questions about refactoring. This study aimed to identify the real challenges developers face when refactoring in the wild and highlight the developers' priorities related to this process. The authors discovered, among other things, that developers mainly ask about design patterns, UI, web services, parallel programming, and mobile apps when discussing refactoring. The findings of this GLR can, for example, help researchers direct their studies towards practical refactoring problems. Although these four previously mentioned studies have conducted SLRs or GLRs on refactoring, none specifically focused on functional languages as our work.

7 Conclusion

This paper proposes and validates a comprehensive catalog of refactorings for Elixir. We used a mixed methodology, based on a systematic literature review, a grey literature review, and mining GitHub code repositories, to catalog 82 refactoring strategies for Elixir systems, 14 of which are novel and specific to the language. In addition to the Elixir-specific refactorings, our catalog comprises 25 traditional refactorings, *i.e.,* described by Fowler and Beck (1999), other 32 refactorings that use characteristic features of functional languages (*e.g.,* pattern matching, list comprehension, pipelines, and higher-order functions), and 11 Erlang-specific transformations, all of which are also compatible with Elixir systems. The detailed documentation of these refactorings and their code examples are available in our GitHub public repository (Vegi and Valente 2023a).

The proposed catalog of refactorings was validated by surveying 144 experienced Elixir developers. These developers, who come from 42 countries spanning all continents, expressed their perceptions regarding the prevalence and relevance of the refactorings that comprise our catalog. The results showed that most refactorings have at least middle relevance, and therefore have a non-negligible potential to improve the quality of Elixir systems. In addition, the use of refactorings during the maintenance and evolution of Elixir systems is not rare, therefore requiring the attention of developers' to learn the motivations and mechanics of these transformation strategies and perform them when opportunities are found.

Future work. As future work, we intend to establish a relationship between the catalog of refactorings for Elixir proposed in this study and the catalog of code smells for this language proposed in our previous work (Vegi and Valente 2023), mapping, for each code smell, the refactoring strategies capable of removing them. Furthermore, we intend to implement or adapt a refactoring tool, such as STYLER, to automatically perform a larger number of refactorings for Elixir.

Replication Package: We provide the complete dataset used in this paper and a replication package at https://doi.org/10.5281/zenodo.11372758. It is composed of (i) links to the papers from the systematic literature review, links to documents from the grey literature review, and links to the artifacts mined from GitHub, thus detailing the respective sources of extraction for each refactoring; (ii) the questions of each survey version; (iii) the survey results in a fully anonymized format; and (iv) the complete results of the Mann-Whitney test.

Appendix - Refactorings not listed in the article's body

To improve the readability of this work, we chose not to show the description of all 82 refactorings cataloged by us in Section 3.2. However, aiming to make this article self-contained, in this appendix we provide descriptions of all refactorings not presented in the article's body. More details about each one, including code examples, are available at https://github.com/lucasvegi/Elixir-Refactorings.

Table 17 Functional Refactorings compatible with Elixir not listed in Section 3.2 (at most two sources)

Refactoring	Description	#
CONVERT NESTED CONDITIONALS TO PIPELINE	Eliminates nested conditionals used only to control a sequence of `function` calls, replacing them with pipe operators. The interfaces of `functions` involved in the pipe are also modified by adding parameters and using pattern matching	2
LIST COMPREHENSION SIMPLIFICATIONS	Transforms a list comprehension (*i.e.,* `for` construct) into semantically equivalent calls to the functions `Enum.map/2` or `Enum.filter/2`	2
REPLACING RECURSION WITH A HIGHER- LEVEL CONSTRUCT	Transforms recursive `functions` into calls to Elixir's built-in higher-order functions (*e.g.,* `Enum.reduce/3`, `Enum.map/2`, etc.)	2
SIMPLIFYING PATTERN MATCHING WITH NESTED STRUCTS	Transforms a pattern matching that performs a deep extraction in nested `structs` into a pattern matching only with the outermost `struct` in the nesting	2
STATIC STRUCTURE REUSE	Eliminates unnecessary recreations of identical static structures (*e.g.,* `lists` or `tuples`) by assigning them to variables that allow these structures to be shared throughout the code	2
STRUCT GUARD TO MATCHING	Transforms calls to `is_struct/1` or `is_struct/2` contained in guards clauses, into explicit pattern matching usage	2

Table 17 continued

Refactoring	Description	#
TRANSFORMING LIST APPENDS AND SUBTRACTS	Transforms calls to the `Enum.concat/2` and `Enum.reject/2` into uses of the `Kernel.++/2` and `Kernel.--/2` operators, respectively	2
MERGING MATCH EXPRESSIONS INTO A LIST PATTERN	Merges a series of match expressions into a single match expression that employs a `list` pattern	1
REPLACE A NESTED CONDITIONAL IN A "CASE" STATEMENT WITH GUARDS	Replaces nested conditional statements (*e.g.,* `if..else`) within a `case` with the use of guards, maintaining the ability to perform more complex pattern matching checks	1
SIMPLIFYING GUARD SEQUENCES	Simplifies a guard clause by eliminating redundancies but preserving the same behavior. For example, we can transform `(is_float(f) and f == 81.0)` into only `(f === 81.0)`	1

#: Number of sources

Table 18 Traditional Refactorings compatible with Elixir not listed in Section 3.2 (at most two sources)

Refactoring	Description	#
INTRODUCE OVERLOADING	Creates a variation of a `function` (*i.e.,* `function` with identical name but different arity or multi-clause `function`), enabling its use in different contexts	2
TRANSFORM "UNLESS" WITH NEGATED CONDITIONS INTO "IF"	Replaces `unless` statements with negated conditions with `if` statements. The reason is because comprehending that something is executed only when a negated condition is not met is confusing	2
GROUP CASE BRANCHES	The branches of a `case` statement in a function is partitioned, replacing the original `case` with separate `case` statements	1
MOVE EXPRESSION OUT OF CASE	Moves an expression outside of a `case` statement when it is repeated at the end of all branches	1
MOVE FILE	Moves a project file containing code such as `modules`, `macros`, `structs`, etc. to another directory, improving the organization of an Elixir project	1
REDUCING A BOOLEAN EQUALITY EXPRESSION	Replaces multiple equality comparisons involving the same variable and logical `OR` operators with the use of the `in` operator and a list containing all possible valid values for the variable	1
REPLACE CONDITIONAL WITH POLYMORPHISM VIA PROTOCOLS	Transforms a `module` that has a `function` with conditionals based on data types into a `Protocol` that defines the interface for this `function`. Furthermore, each data type previously handled in the conditionals is converted into a specific implementation of the `Protocol`	1
SIMPLIFYING CHECKS BY USING TRUTHNESS CONDITION	Replaces an `if..else`, that checks an `is_nil/1` call used to return a default value if a given data is indeed null (*e.g.,* `if is_nil(data), do: "default", else: data`), with a short-circuit operator based on truthness conditions (*e.g.,* `data || "default"`)	1

#: Number of sources

Acknowledgements We thank the Elixir developers who participated in our survey and shared their perspectives about refactoring in Elixir. Finally, 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).

Funding This research is supported by a grant from Dashbit: https://dashbit.co/ and Rebase: https://rebase.com.br/.

Declarations

Compliance with Ethical Standards The authors declare that they have no conflict of interest. Since this research involved the active participation of human subjects, the survey was evaluated and approved by the Research Ethics Committee at the Federal University of Minas Gerais, Brazil (CAAE: 76454823.8.0000.5149) before its execution. The questionnaire began by presenting the study's objectives, introducing the research team, explaining the voluntary nature of participation, and obtaining participants' consent.

References

Abid C, Gaaloul K, Kessentini M, Alizadeh V (2022) What refactoring topics do developers discuss? a large scale empirical study using Stack Overflow. IEEE Access 10:56362–56374. https://doi.org/10.1109/ACCESS.2021.3140036

Abid C, Alizadeh V, Kessentini M, Ferreira TN, Dig D (2020) 30 years of software refactoring research: a systematic literature review 1–23. https://doi.org/10.48550/arXiv.2007.02194

Al Dallal J, Abdin A (2018) Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: a systematic literature review. IEEE Trans Softw Eng 44(1):44–69. https://doi.org/10.1109/TSE.2017.2658573

Almeida U (2018) Learn functional programming with Elixir: new foundations for a new world, 1 edn. Pragmatic Bookshelf

Arnaoudova V, Constantinides C (2008) Adaptation of refactoring strategies to multiple axes of modularity: Characteristics and criteria. In: 6th International conference on software engineering research, management and applications (SERA), pp 105–114. https://doi.org/10.1109/SERA.2008.38

Avgerinos T, Sagonas K (2009) Cleaning up Erlang code is a dirty job but somebody's gotta do it. In: 8th ACM SIGPLAN Workshop on ERLANG, pp 1–10. https://doi.org/10.1145/1596600.1596602

Backfield J (2014) Becoming functional: steps for transforming into a functional programmer, 1 edn. O'Reilly Media

Barn B, Barat S, Clark T (2017) Conducting systematic literature reviews and systematic mapping studies. In: 10th Innovations in software engineering conference (ISEC), pp 212-213. https://doi.org/10.1145/3021460.3021489

Barwell AD, Brown CM, Hammond K (2018) Finding parallel functional pearls: automatic parallel recursion scheme detection in Haskell functions via anti-unification. Future Gener Comput Syst 79:669–686. https://doi.org/10.1016/j.future.2017.07.024

Bellegarde F (1995) Notes for pipelines of transformations for ML. Oregon Graduate Institute of Science & Technology, Technical report

Bibiano AC, Soares V, Coutinho D, Fernandes E, Correia JaL, Santos K, Oliveira A, Garcia A, Gheyi R, Fonseca B, Ribeiro M, Barbosa C, Oliveira D (2020) How does incomplete composite refactoring affect internal quality attributes? In: 28th International conference on program comprehension (ICPC), pp 149-159. https://doi.org/10.1145/3387904.3389264

Bordignon MD, Silva RA (2020) Mutation operators for concurrent programs in Elixir. In: 21st IEEE Latin-American test symposium (LATS), pp 1–6. https://doi.org/10.1109/LATS49555.2020.9093675

Bozó I, Fördős V, Horpácsi D, Horváth Z, Kozsik T, Kőszegi J, Tóth M (2015) Refactorings to enable parallelization. In: 15th Trends in functional programming (TFP), pp 104–121. https://doi.org/10.1007/978-3-319-14675-1_7

Bozó I, Fordós V, Horvath Z, Tóth M, Horpácsi D, Kozsik T, Köszegi J, Barwell A, Brown C, Hammond K (2014) Discovering parallel pattern candidates in Erlang. In: 13th ACM SIGPLAN Workshop on Erlang, pp 13-23. https://doi.org/10.1145/2633448.2633453

Bozó I, Tóth xM (2009) Restructuring Erlang programs using function related refactorings. In: 11th Symposium on programming languages and software tools and 7th nordic workshop on model driven software engineering (SPLST & NW-MODE), pp 162–176

Brito A, Hora A, Valente MT (2024) Towards a catalog of composite refactorings. J Softw Evol Process 36(4):1–22. https://doi.org/10.1002/smr.2530
Brown CM, Hammond K, Danelutto M, Kilpatrick P, Schöner H, Breddin T (2011) Paraphrasing: generating parallel programs using refactoring. In: 10th Symposium on Formal Methods for Components and Objects (FMCO), pp 237–256. https://doi.org/10.1007/978-3-642-35887-6_13
Brown CM, Li H, Thompson S (2011) An expression processor: a case study in refactoring Haskell programs. In: Page R, Horváth Z, Zsók V (eds) 12th Trends in functional programming (TFP), vol 6546, pp 31–49. https://doi.org/10.1007/978-3-642-22941-1_3
Brown CM, Loidl HW, Hammond K (2011) Paraforming: forming parallel Haskell programs using novel refactoring techniques. In: 12th International conference on trends in functional programming (TPF), pp 82–97. https://doi.org/10.1007/978-3-642-32037-8_6
Brown CM (2008) Tool support for refactoring Haskell programs. Phd thesis, University of Kent, UK
Brown CM, Danelutto M, Hammond K, Kilpatrick P, Elliott A (2014) Cost-directed refactoring for parallel Erlang programs. Int J Parall Program 42:564–582. https://doi.org/10.1007/s10766-013-0266-5
Budgen D, Brereton P (2006) Performing systematic literature reviews in software engineering. In: 28th International conference on software engineering (ICSE), pp 1051–1052. https://doi.org/10.1145/1134285.1134500
Burstall RM, Darlington J (1977) A transformation system for developing recursive programs. J ACM 24(1):44–67. https://doi.org/10.1145/321992.321996
Chang S (2013) Laziness by need. In: Felleisen M, Gardner P (eds) Programming languages and systems (ESOP), pp 81–100. https://doi.org/10.1007/978-3-642-37036-6_5
Chechina N, MacKenzie K, Thompson S, Trinder P, Boudeville O, Fördős V, Hoch C, Ghaffari A, Hernandez MM (2017) Evaluating scalable distributed Erlang for scalability and reliability. IEEE Trans Parall Distrib Syst 28(8):2244–2257. https://doi.org/10.1109/TPDS.2017.2654246
Corbat T, Felber L, Stocker M, Sommerlad P (2007) Ruby refactoring plug-in for Eclipse. In: 22nd ACM SIGPLAN conference on object-oriented programming systems and applications companion (OOPSLA), pp 779–780. https://doi.org/10.1145/1297846.1297884
Cruzes DS, Dybå T (2011a) Recommended steps for thematic synthesis in software engineering. In: 5th International symposium on empirical software engineering and measurement (ESEM), pp 275–284. https://doi.org/10.1109/ESEM.2011.36
Cruzes DS, Dybå T (2011b) Research synthesis in software engineering: a tertiary study. Inf Softw Technol 53(5):440–455. https://doi.org/10.1016/j.infsof.2011.01.004
Dabic O, Aghajani E, Bavota G (2021) Sampling projects in Github for MSR studies. In: 18th IEEE/ACM international conference on mining software repositories (MSR), pp 560–564. https://doi.org/10.1109/MSR52588.2021.00074
Drienyovszky D, Horpácsi D, Thompson S (2010) Quickchecking refactoring tools. In: 9th ACM SIGPLAN Workshop on Erlang, pp 75-80. https://doi.org/10.1145/1863509.1863521
Erwig M, Ren D (2007) An update calculus for expressing type-safe program updates. Sci Comput Program 67(2):199–222. https://doi.org/10.1016/j.scico.2007.01.003
Farmer A, Gill A, Komp E, Sculthorpe N (2012) The HERMIT in the machine: a plugin for the interactive transformation of GHC core language programs. In: ACM SIGPLAN symposium on haskell (Haskell'12), pp 1–12. https://doi.org/10.1145/2364506.2364508
Farmer A, Sculthorpe N, Gill A (2015) Reasoning with the HERMIT: tool support for equational reasoning on GHC core programs. In: ACM SIGPLAN Symposium on Haskell (Haskell'15), pp 23–34. https://doi.org/10.1145/2804302.2804303
Fowler M, Beck K (1999) Refactoring: improving the design of existing code, 1 edn. Addison-Wesley
Garousi V, Felderer M, Mäntylä MV (2019) Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. Inf Softw Technol 106(1):101–121. https://doi.org/10.1016/j.infsof.2018.09.006
Gill A (2006) Introducing the Haskell equational reasoning assistant. In: ACM SIGPLAN workshop on Haskell (Haskell'06), pp 108–109. https://doi.org/10.1145/1159842.1159856
Golubev Y, Kurbatova Z, AlOmar EA, Bryksin T, Mkaouer MW (2021) One thousand and one stories: a large-scale survey of software refactoring. In: 29th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering (ESEC/FSE), pp 1303–1313. https://doi.org/10.1145/3468264.3473924
Guttmann W, Partsch H, Schulte W, Vullinghs T (2003) Tool support for the interactive derivation of formally correct functional programs. J Univ Comput Sci 9(2):173–188. https://doi.org/10.3217/jucs-009-02-0173
Héder M, László Z, Sulyán T (2007) Another neat tool for refactoring Erlang programs. In: 11th IASTED international conference on software engineering and applications (SEA), pp 336–341

Hudak P (1989) Conception, evolution, and application of functional programming languages. ACM Comput Surv 21(3):359–411. https://doi.org/10.1145/72551.72554
Janjic V, Brown C, Barwell A, Hammond K (2021) Refactoring for introducing and tuning parallelism for heterogeneous multicore machines in Erlang. Concurr Comput Pract Exper 33(14):1–25
Johnsson T (1985) Lambda lifting: transforming programs to recursive equations. In: Functional programming languages and computer architecture (FPCA), pp 190–203. https://doi.org/10.1007/3-540-15975-4_37
Jurić S (2024) Elixir in action, 3 edn. Manning
Kamei F, Wiese I, Lima C, Polato I, Nepomuceno V, Ferreira W, Ribeiro M, Pena C, Cartaxo B, Pinto G, Soares S (2021) Grey literature in software engineering: a critical review. Inf Softw Technol 138(1):1–26. https://doi.org/10.1016/j.infsof.2021.106609
Király R (2013) Complexity metric based source code transformation of Erlang programs. In: Annales Mathematicae et Informaticae, pp 29–44
Kitchenham B, Charters S (2007) Guidelines for performing systematic literature reviews in software engineering. Technical report, Keele University
Kozsik T, Tóth M, Bozó I, Horváth Z (2017) Static analysis for divide-and-conquer pattern discovery. Comput Inform 35(4):764–791
Kozsik T, Tóth M, Bozó I (2018) Free the conqueror! refactoring divide-and-conquer functions. Future Gener Comput Syst 79:687–699. https://doi.org/10.1016/j.future.2017.05.011
Kozsik T, Csörnyei Z, Horváth Z, Király R, Kitlei R, Lövei L, Nagy T, Tóth M, Víg A (2007) Use cases for refactoring in Erlang. In: 2nd Central european functional programming school (CEFP), pp 250–285. https://doi.org/10.1007/978-3-540-88059-2_7
Lämmel R (2002) Towards generic refactoring. In: ACM SIGPLAN workshop on Rule-Based Programming (RULE), pp 15-28. https://doi.org/10.1145/570186.570188
Lee DY (2011) A case study on refactoring in Haskell programs. In: 33rd International conference on software engineering (ICSE), pp 1164–1166. https://doi.org/10.1145/1985793.1986030
Li H, Thompson S, Reinke C (2005) The Haskell refactorer, HaRe, and its API. Electron Notes Theor Comput Sci 141(4):29–34. https://doi.org/10.1016/j.entcs.2005.02.053
Li H, Thompson S (2006) Comparative study of refactoring Haskell and Erlang programs. In: 6th IEEE international workshop on source code analysis and manipulation (SCAM), pp 197–206. https://doi.org/10.1109/SCAM.2006.8
Li H, Thompson S (2012) A domain-specific language for scripting refactorings in Erlang. In: de Lara J, Zisman A (eds) Fundamental approaches to software engineering (FASE), vol 7212, pp 501–515. https://doi.org/10.1007/978-3-642-28872-2_34
Li H, Thompson S (2015) Safe concurrency introduction through slicing. In: Workshop on partial evaluation and program manipulation (PEPM), pp 103-113. https://doi.org/10.1145/2678015.2682533
Li H, Thompson S, Lamela Seijas P, Francisco MA (2014) Automating property-based testing of evolving web services. In: ACM SIGPLAN Workshop on partial evaluation and program manipulation (PEPM), pp 169-180. https://doi.org/10.1145/2543728.2543741
Li H, Thompson S, Lövei L, Horváth Z, Kozsik T, Víg A, Nagy T (2006) Refactoring Erlang programs. In: 12th International Erlang/OTP user conference (EUC), pp 1–10
Li H, Thompson S, Orosz G, Tóth M (2008) Refactoring with Wrangler, updated: data and process refactorings, and integration with Eclipse. In: 7th ACM SIGPLAN Workshop on ERLANG, pp 61–72. https://doi.org/10.1145/1411273.1411283
Lövei L, Hoch C, Köllö H, Nagy T, Nagyné Víg A, Horpácsi D, Kitlei R, Király R (2008) Refactoring module structure. In: 7th ACM SIGPLAN Workshop on ERLANG, pp 83–89. https://doi.org/10.1145/1411273.1411285
Lövei L, Horváth Z, Kozsik T, Király R (2007a) Introducing records by refactoring in Erlang programs. In: 10th Symposium on programming languages and software tools (SPLST), pp 1–18
Lövei L, Horváth Z, Kozsik T, Király R (2007b) Introducing records by refactoring. In: SIGPLAN Workshop on ERLANG Workshop, pp 18–28. https://doi.org/10.1145/1292520.1292524
Murphy-Hill E, Parnin C, Black AP (2012) How we refactor, and how we know it. IEEE Trans Softw Eng 38(1):5–18. https://doi.org/10.1109/TSE.2011.41
Nardone V, Muse BA, Abidi M, Khomh F, Penta MD (2023) Video game bad smells: What they are and how developers perceive them. ACM Trans Softw Eng Methodol 32(4):1–35. https://doi.org/10.1145/3563214
Oliveira J, Gheyi R, Mongiovi M, Soares G, Ribeiro M, Garcia A (2019) Revisiting the refactoring mechanics. Inf Softw Technol 110:136–138. https://doi.org/10.1016/j.infsof.2019.03.002
Opdyke WF (1992) Refactoring object-oriented frameworks. Phd thesis, University of Illinois at Urbana-Champaign, USA

Page L, Brin S, Motwani R, Winograd T (1999) The PageRank citation ranking: bringing order to the Web. Technical report, Stanford InfoLab
Partsch H, Steinbrüggen R (1983) Program transformation systems. ACM Comput Surv 15(3):199–236. https://doi.org/10.1145/356914.356917
Ratzinger J, Sigmund T, Gall HC (2008) On the relation of refactorings and software defect prediction. In: International working conference on mining software repositories (MSR), pp 35-38. https://doi.org/10.1145/1370750.1370759
Rowe RNS, Férée H, Thompson S, Owens S (2019a) Characterising renaming within OCaml's module system: theory and implementation. In: 40th ACM SIGPLAN conference on programming language design and implementation (PLDI), pp 950–965. https://doi.org/10.1145/3314221.3314600
Rowe RNS, Férée H, Thompson S, Owens S (2019b) ROTOR: a tool for renaming values in OCaml's module system. In: 3rd International workshop on refactoring (IWOR), pp 27–30. https://doi.org/10.1109/IWoR.2019.00013
Sagonas K, Avgerinos T (2009) Automatic refactoring of Erlang programs. In: 11th ACM SIGPLAN conference on principles and practice of declarative programming (PPDP), pp 13–24. https://doi.org/10.1145/1599410.1599414
Sagonas K, Luna D (2008) Gradual typing of Erlang programs: a wrangler experience. In: 7th ACM SIGPLAN Workshop on ERLANG, pp 73–82. https://doi.org/10.1145/1411273.1411284
Scott ML (2015) Programming Language Pragmatics, 4 edn. Morgan Kaufmann
Sculthorpe N, Farmer A, Gill A (2013) The HERMIT in the tree - mechanizing program transformations in the GHC core language. In: Hinze R (ed) Symposium on implementation and application of functional languages (IFL). Lecture Notes in Computer Science, vol 8241, pp 86–103
Sherwany A, Zaza N, Nystrom N (2015) A refactoring library for Scala compiler extensions. In: Franke B (ed) 24th International conference on compiler construction (CC). Lecture notes in computer science, vol 9031, pp 31–48. https://doi.org/10.1007/978-3-662-46663-6_2
Siegel S, Castellan J (1988) Nonparametric Statistics for the Behavioral Sciences, 2 edn. McGraw-Hill International Editions
Silva D, Tsantalis N, Valente MT (2016) Why we refactor? confessions of Github contributors. In: 24th ACM SIGSOFT international symposium on foundations of software engineering (FSE), pp 858–870. https://doi.org/10.1145/2950290.2950305
Singh S, Kaur S (2018) A systematic literature review: refactoring for disclosing code smells in object oriented software. Ain Shams Eng J 9(4):2129–2151. https://doi.org/10.1016/j.asej.2017.03.002
Taylor R, Derrick J (2015) mu2: a refactoring-based mutation testing framework for Erlang. In: 27th International conference on testing software and systems (ICTSS), pp 178–193. https://doi.org/10.1007/978-3-319-25945-1_11
Thomas D (2018) Programming Elixir |> 1.6: functional |> concurrent |> pragmatic |> fun, 1 edn. Pragmatic Bookshelf
Thompson S (2005) Refactoring functional programs. In: 5th International summer school on advanced functional programming (AFP), pp 331–357. https://doi.org/10.1007/11546382_9
Thompson S, Horpacsi D, Koszegi J (2016) Towards trustworthy refactoring in Erlang. In: 4th International workshop on verification and program transformation (VPT), pp 83–103
Thompson S, Reinke C (2003) A case study in refactoring functional programs. In: 7th Brazilian symposium on programming languages (SBLP), pp 1–16
Tóth M, Bozó I, Kozsik T (2017) Pattern candidate discovery and parallelization techniques. In: 29th Symposium on the implementation and application of functional programming languages (IFL), pp 1–26. https://doi.org/10.1145/3205368.3205369
Tullsen MA (2002) PATH, a program transformation system for Haskell. Phd thesis, Yale University, USA
Vegi LFM, Valente MT (2022) Code smells in Elixir: early results from a grey literature review. In: 30th International conference on program comprehension (ICPC), pp 580–584. https://doi.org/10.1145/3524610.3527881
Vegi LFM, Valente MT (2023a) Catalog of Elixir refactorings. https://github.com/lucasvegi/Elixir-Refactorings
Vegi LFM, Valente MT (2023b) Towards a catalog of refactorings for Elixir. In: 39th International conference on software maintenance and evolution (ICSME), pp 358–362. https://doi.org/10.1109/ICSME58846.2023.00045
Vegi LFM, Valente MT (2024) Understanding refactorings in Elixir functional language - Replication Package. https://doi.org/10.5281/zenodo.11372758
Vegi LFM, Valente MT (2023) Understanding code smells in Elixir functional language. Empir Softw Eng 28(102):1–32. https://doi.org/10.1007/s10664-023-10343-6

Villavicencio G (2011) A bottom-up approach to understand functional programs. In: 4th International C* conference on computer science and software engineering (C3S2E), pp 111-120. https://doi.org/10.1145/1992896.1992910
Walpole RE, Myers RH, Myers SL, Ye K (2007) Probability & statistics for engineers and scientists, 8 edn. Pearson Education
Weißgerber P, Diehl S (2006) Are refactorings less error-prone than other changes? In: International workshop on mining software repositories (MSR), pp 112-118. https://doi.org/10.1145/1137983.1138011
Wohlin C (2014) Guidelines for snowballing in systematic literature studies and a replication in software engineering. In: 18th International conference on evaluation and assessment in software engineering (EASE), pp 1–10. https://doi.org/10.1145/2601248.2601268
Xing Z, Stroulia E (2006) Refactoring practice: how it is and how it should be supported - an Eclipse case study. In: 22nd IEEE international conference on software maintenance (ICSM), pp 458–468. https://doi.org/10.1109/ICSM.2006.52
Zhang Z, Xing Z, Xia X, Xu X, Zhu L (2022) Making python code idiomatic by automatic refactoring non-idiomatic python code with pythonic idioms. In: 30th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering (ESEC/FSE), pp 696-708. https://doi.org/10.1145/3540250.3549143
Zhang H, Zhou X, Huang X, Huang H, Babar MA (2020) An evidence-based inquiry into the use of grey literature in software engineering. In: 42nd ACM/IEEE international conference on software engineering (ICSE), pp 1422-1434. https://doi.org/10.1145/3377811.3380336