UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

Lucas Francisco da Matta Vegi

Code Smells and Refactorings for Elixir

Belo Horizonte
2024

Lucas Francisco da Matta Vegi

Code Smells and Refactorings for Elixir

Final Version

Thesis presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Doctor in Computer Science.

Advisor: Marco Túlio de Oliveira Valente

Belo Horizonte
2024

Vegi, Lucas Francisco da Matta.

V423c Code smells and refactorings for Elixir [recurso eletrônico] / Lucas Francisco da Matta Vegi – 2024.
1 recurso online (222 f. il, color.) : pdf.

Orientador: Marco Túlio de Oliveira Valente.

Tese (Doutorado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação.
Referências: f.147- 167

1. Computação – Teses. 2. Engenharia de software – Teses. 3. Elixir (Linguagem de programação de computadores) – Teses. 4. Refatoração de software – Teses. 5. Code smells – Teses I. Valente, Marco Túlio de Oliveira. II. Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Computação. III.Título.

CDU 519.6*32(043)

Ficha catalográfica elaborada pela bibliotecária Irenquer Vismeg Lucas Cruz
CRB 6/819 - Universidade Federal de Minas Gerais - ICEx

UNIVERSIDADE FEDERAL DE MINAS GERAIS

CODE SMELLS AND REFACTORINGS FOR ELIXIR

LUCAS FRANCISCO DA MATTA VEGI

Tese defendida e aprovada pela banca examinadora constituída pelos Senhores:

Prof. Marco Túlio de Oliveira Valente - Orientador
Departamento de Ciência da Computação - UFMG

Prof. Rohit Gheyi
Departamento de Sistemas e Computação - UFCG

Prof. Marcelo de Almeida Maia
Faculdade de Computação - UFU

Prof. André Cavalcante Hora
Departamento de Ciência da Computação - UFMG

Prof. Eduardo Magno Lages Figueiredo
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 10 de dezembro de 2024.

Documento assinado eletronicamente por **Marco Tulio de Oliveira Valente**, **Professor do Magistério Superior**, em 06/02/2025, às 12:03, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Andre Cavalcante Hora**, **Professor do Magistério Superior**, em 06/02/2025, às 19:58, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Eduardo Magno Lages Figueiredo**, **Professor do Magistério Superior**, em 07/02/2025, às 02:15, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Marcelo de Almeida Maia**, **Usuário Externo**, em

10/02/2025, às 10:00, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Rohit Gheyi**, **Usuário Externo**, em 18/02/2025, às 09:48, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

A autenticidade deste documento pode ser conferida no site https://sei.ufmg.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **3947797** e o código CRC **B616CD36**.

Referência: Processo nº 23072.208285/2025-43 SEI nº 3947797

To José and Aline, my beloved son and dear wife.

Acknowledgments

Esta tese é fruto de uma longa jornada de entrega, aprendizado e superação pessoal. Não seria possível construir um caminho como esse sozinho. Por esse motivo, gostaria de agradecer a todos que, de alguma forma, contribuíram nesse período importante da minha vida. Sob o risco de esquecer figuras essenciais, agradeço, em especial:

A Deus, por me dar força em todos os momentos.

Aos meus pais e avós, que me ensinaram valores inegociáveis.

Aos meus primos/padrinhos Alexandre e Ana, que acolheram a mim e à minha esposa durante todo o tempo que residimos em Belo Horizonte.

À minha esposa, Aline, que, além de ser a minha grande inspiração para seguir na carreira acadêmica, abriu mão do seu sonho para viabilizar o meu.

Ao professor José Luis Braga (UFV), principal responsável pela minha paixão pela área de Engenharia de Software. Além disso, por meio de seu incentivo, pude conhecer meu orientador, dando início, assim, a essa jornada.

Ao meu orientador, Marco Túlio Valente, por todo o respeito, atenção, empatia e ensinamentos. Foi um enorme prazer ter a oportunidade de trabalhar com uma pessoa tão especial em tantos aspectos. Tenho certeza de que a parceria continuará.

À comunidade Elixir, pela acolhida e pelas participações abundantes ao longo das diversas etapas deste trabalho. Ao professor Adolfo Neto (UTFPR), faço um agradecimento mais especial, pois, além de ter despertado meu interesse por essa linguagem por meio de seus podcasts, ele também foi um dos grandes responsáveis por tornar a mim e ao meu trabalho conhecidos por essa comunidade de desenvolvedores.

A José Valim, criador do Elixir, por facilitar nossa comunicação com a comunidade Elixir e intermediar o suporte financeiro fornecido a esta pesquisa pelas empresas FinBits, Dashbit e Rebase, a quem também agradeço.

Aos membros da banca, pela disponibilidade em contribuir com este trabalho.

Ao DPI/UFV, por me conceder licença integral, permitindo-me manter o foco exclusivamente na pesquisa nesses últimos anos.

Ao DCC/UFMG, pelo suporte financeiro, logístico e profissional.

"*Have a healthy disregard for the impossible.*"
(Larry Page)

Resumo

Elixir é uma linguagem de programação funcional moderna, criada em 2012, cuja popularidade vem crescendo na indústria. No entanto, até onde sabemos, existem poucos estudos na literatura que abordem a qualidade interna de sistemas implementados com essa linguagem. Particularmente, nenhum estudo até o momento investigou *code smells* ou refatorações específicos para Elixir. Portanto, para aproveitar essas oportunidades de pesquisa, nos inspiramos no conhecido livro de Fowler sobre *code smells* e refatorações para prospectar, estudar, documentar e avaliar *code smells* e estratégias de refatoração adaptadas especificamente a Elixir. Em um primeiro estudo, utilizamos uma abordagem metodológica mista para catalogar 35 *code smells*, dos quais 23 são novos e específicos para Elixir, enquanto 12 são *code smells* tradicionais catalogados por Fowler e Beck, que também afetam códigos implementados nessa linguagem. Esse catálogo foi validado por meio de questionários respondidos por 181 desenvolvedores experientes em Elixir, oriundos de 37 países e de todos os continentes. Em um segundo estudo, também adotamos uma abordagem metodológica mista, que incluiu uma revisão sistemática da literatura, para catalogar 82 estratégias de refatoração compatíveis com Elixir, sendo 14 delas inéditas e específicas para essa linguagem. Todas essas refatorações foram validadas por meio de novos questionários respondidos por 151 desenvolvedores de 42 países diferentes. Para documentar os *code smells* e as refatorações catalogadas, além de descrições textuais estruturadas, produzimos exemplos de código que os representam. Por fim, conduzimos um terceiro estudo, no qual os *code smells* e as refatorações para Elixir foram correlacionados, permitindo assim a definição de diretrizes práticas sobre como cada *code smell* pode ser removido de forma disciplinada com a ajuda de estratégias de refatoração. Nesse último estudo, também catalogamos cinco novas refatorações compostas para Elixir. De maneira geral, os resultados desta tese têm implicações práticas relacionadas à prevenção e remoção de *code smells* em Elixir, bem como à priorização do entendimento e uso das estratégias de refatoração para essa linguagem.

Palavras-chave: code smells; refatoração; Elixir; programação funcional; mineração de repositórios de software; revisão da literatura cinza; revisão sistemática da literatura.

Abstract

Elixir is a modern functional programming language, created in 2012, whose popularity has been growing in the industry. Despite this fact, and to the best of our knowledge, there are few studies in the literature addressing the internal quality of systems implemented with this language. In particular, no study to date has investigated specific code smells or refactorings for Elixir. Therefore, to fill these research gaps, we take inspiration from Fowler's well-known book on code smells and refactorings to prospect, study, document, and evaluate code smells and refactoring strategies specifically tailored to Elixir. In the first study, we employed a mixed-method approach to catalog 35 code smells, 23 of which are new and specific to Elixir, while 12 are traditional code smells cataloged by Fowler and Beck that also affect code implemented in this language. We validated this catalog by surveying 181 experienced Elixir developers from 37 countries across all continents. In a second study, we also adopted a mixed-method approach, including a systematic literature review, to catalog 82 refactoring strategies compatible with Elixir, 14 of which are novel and specific to this language. All these refactorings were validated through another survey answered by 151 developers from 42 countries. To document the cataloged code smells and refactorings, in addition to structured textual descriptions, we produced code examples representing them. Finally, we conducted a third study where the code smells and refactorings for Elixir were correlated, allowing the definition of practical guidelines on how each code smell can be removed systematically with the help of refactoring strategies. In this final study, we also cataloged five new composite refactorings for Elixir. Overall, the results of this thesis have practical implications for the prevention and removal of code smells in Elixir, as well as the prioritization of understanding and using refactoring strategies for this language.

Keywords: code smells; refactoring; Elixir; functional programming; mining software repositories; grey literature review; systematic literature review.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

This chapter introduces this Ph.D. thesis. We begin by presenting our problem and motivation in Section 1.1. Next, in Section 1.2, we detail the objectives, goals, and major contributions of this work. In Section 1.3, we list our current publications. Finally, we present the outline of this thesis in Section 1.4.

1.1 Problem and Motivation

The concern for ensuring product quality is a common characteristic among engineering disciplines, and software engineering is no exception. According to Meyer [128], the assessment of software quality can be classified into two categories: *external quality* and *internal quality*. Based on this definition, the external quality of software measures quality attributes that do not depend on source code to be evaluated, such as robustness, correctness, usability, and efficiency. On the other hand, the internal quality of software evaluates quality attributes directly related to its code, such as maintainability, testability, and readability.

According to Valente [193], successive maintenance activities over time lead to an increase in complexity and difficulty in maintaining the code and internal structure of a system. Essentially, there is a progressive degradation of the system's internal quality as maintenance interventions and enhancements occur. This fact is also mentioned by Fowler in the preface of his well-known book on refactoring [74], where he recounts an experience as a consultant on a project that failed for this reason: *"...the project failed, in large part because the code [became] too complex to debug or to tune to acceptable performance"*. This same failed project was successfully restarted and maintained by applying refactoring techniques, which are code transformations aimed at improving the quality of a system without altering its behavior [74], thereby stabilizing the natural decline in quality of these systems after successive maintenance activities [193].

The success of this project motivated Fowler to write his aforementioned book on

refactorings [74] to communicate to other developers how to improve the quality of their code. In this book, Fowler cataloged 72 refactorings for object-oriented code, helping to popularize these code transformation techniques. Additionally, Fowler and Beck coined the term *"code smell"* to describe sub-optimal code structures that can harm the evolution of software, characterizing them as opportunities for refactoring. In total, Fowler's book cataloged 22 code smells, also contributing to the popularization of this concept.

Since the impacts on software quality caused by code smells are not homogeneous and can vary across different domains [72], developers' perception of code smells can also differ [181]. Particularly, each programming language has its own constraints and challenges to perform refactorings [106], and thus there is vast research on code smells and refactoring strategies for specific contexts and languages, such as mobile applications [82, 83], JavaScript [63, 67, 155], CSS [146], games [27, 133], Java [57], Python [215, 216], Ruby [11, 49], and quantum computing [46, 217]. However, the majority of these studies are focused on improving the quality of object-oriented systems [1, 176].

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 [26]. More specifically, Elixir is a modern functional programming language that is gaining traction in the industry, with over 300 companies worldwide using the language, including Discord, Heroku, and PepsiCo.[1] This language is renowned for its performance in parallel and distributed computing environments [183]. Conceived in 2012, Elixir draws inspiration from a blend of programming languages, such as Ruby, Haskell, Erlang, and Clojure [94]. According to approximately 72% of developers who participated in the Elixir Survey 2023,[2] the three main factors that influenced their decisions to adopt Elixir are its functional paradigm, increased productivity, and facilitated support for concurrency.

The recent results of the Stack Overflow Survey[3] also highlight how Elixir is a relevant language in the industry today. According to the last three editions of this survey (*i.e.,* 2022, 2023, and 2024), Elixir is the second most admired programming language among developers, just after Rust. This suggests that a large number of developers who currently use the language intend to continue using it in the coming years. Additionally, according to these surveys, Phoenix[4]—the main framework for web development with Elixir—is currently the most loved web technology. Finally, the Stack Overflow Survey 2024 shows that Elixir developers are the second highest-paid in the industry, only behind Erlang developers—the language that most influenced Elixir's design [183].

Although Elixir is becoming particularly popular and relevant in the industry, to the best of our knowledge, no study has yet investigated code smells or refactorings specifically tailored for this language. However, just as in any programming language, **it is**

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

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

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

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

natural to expect that Elixir developers will make bad design choices and then implement sub-optimal code structures, making their systems challenging to maintain, comprehend, modify, and test. Thus, we also expect these developers to pursue design improvements within their codebase through refactoring strategies.

Therefore, considering the growing significance of functional languages, particularly Elixir, and the lack of studies regarding the quality of systems developed with this language, this thesis aims to fill this research gap.

1.2 Objectives and Contributions

As previously mentioned, there is a lack of studies examining design and maintenance issues that are specific to systems implemented using Elixir. The main reason for this gap is that existing literature predominantly focuses on the object-oriented paradigm and more traditional programming languages. Therefore, taking inspiration from Fowler's book [74] but applied in a specific context, **the general objective of this Ph.D. thesis is described as follows:**

> We aim to prospect, study, document, evaluate, and correlate code smells and refactoring strategies specifically tailored to the Elixir functional language.

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

1. First, we cataloged specific code smells for the Elixir language by extracting these sub-optimal structures from multiple content sources, such as grey literature documents, open-source project codebases, and direct interactions with developers of the language. These code smells were validated by developers who work with Elixir.

2. In the second working unit, we cataloged refactoring strategies specifically aimed at improving the quality of systems developed in Elixir, and we also validated these strategies with developers who work with the language. For each of these strategies, we provided examples showing the code before and after the transformations, as well as some conditions to help preserve the behavior of the refactored code.

3. Finally, in the third working unit, we established relationships between the catalogs of code smells and refactorings specific to systems developed in Elixir. This mapping enables us to provide practical guidance to developers on which refactoring strategies

can be used to remove each code smell in a disciplined way, thereby improving the quality of the code.

We summarize each work and highlight their contribution in the remainder of this section. In addition to presenting these scientific contributions, at the end of this section, we also list some indirect contributions from this thesis that are already having an impact on the Elixir developer community.

1.2.1 Code Smells in Elixir

In the first working unit of this thesis, aiming to build a catalog of code smell for Elixir, we *first* conducted a qualitative study, extracting and cataloging code smells for Elixir from 17 grey literature documents, 25 documents created by interactions with the Elixir community in GitHub (13 issues and 12 pull requests), and 46 artifacts mined from Elixir repositories on GitHub. *Second*, we conducted a survey that collected quantitative data with 181 Elixir developers. Each participant received a list of smells and they were asked to rank each one's relevance and prevalence on a scale of one (*very low*) to five (*very high*). In this study, we present the following contributions:

- We cataloged 23 novel Elixir-specific code smells and categorize them into two different groups (Low-Level Concerns smells and Design-Related smells).
- We find that at least 12 traditional code smells (as proposed by Fowler and Beck [74]) are also present in Elixir systems.
- We showed through the results of a survey that the majority of cataloged smells (97%) have at least mid-relevance levels, therefore having the potential to impair the readability, maintenance, or evolution of Elixir systems. Additionally, most smells (54%) have at least mid-prevalence levels, making them common in production code.

These findings have practical implications for developers and researchers, such as establishing priorities for preventing and removing code smells, directing efforts toward the evolution of tools for the automatic detection of code smells in Elixir, and identifying open research fields related to the catalog.

1.2.2 Refactorings in Elixir

In the second working unit, with the goal of cataloging refactoring strategies tailored for Elixir, we *first* conducted a systematic literature review where we analyzed 135 research papers to identify and catalog refactoring strategies that have been proposed for other functional languages but that are also compatible with Elixir. *Second*, we cataloged refactorings for Elixir from 26 grey literature documents. *Third*, we mined 119 artifacts from the Top-10 Elixir repositories with the most stars on GitHub to expand our catalog of refactorings. *Fourth*, we surveyed 144 experienced Elixir developers. Each participant was given a list of refactorings and asked to rate the relevance and prevalence of each on a scale from one (*very low*) to five (*very high*). With this study we achieved the following contributions:

- We cataloged 82 refactorings for Elixir and categorized them into four different groups (Elixir-Specific refactorings, Functional refactorings, Erlang-Specific refactorings, and Traditional refactorings).
- We provided documentation with code examples and some tailored side conditions that can support the implementation of automated refactoring tools for Elixir in the future.
- 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 Elixir systems. 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.
- 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, when developers are learning the refactoring strategies we cataloged, they should prioritize mastering the most prevalent ones first, as understanding these transformations can save time during code reviews. Conversely, when maintaining their systems and encountering multiple refactoring opportunities, developers should apply the most relevant refactorings first to maximize code quality improvements.

1.2.3 Relationship between Code Smells and Refactorings in Elixir

In previous working units, we have cataloged code smells and refactorings specifically tailored for Elixir, but we did not establish direct correlations between them. Therefore, intending to correlate these catalogs in the same way Fowler and Beck [74] did for theirs, and thus to create a practical guide on how to remove each code smell in a disciplined way, in this final working unit we *first* conducted an empirical study where each of the 35 code smells previously cataloged by us was manually compared with each of the 82 refactorings. Through these comparisons, we identified the refactorings that could aid in removing each smell in Elixir and the order in which they should be performed. *Second*, we classified the motivations behind each refactoring not mapped to removing Elixir smells, aiming to understand the reasons for these mapping absences. In this context, we provide the following contributions:

- We found that all 35 code smells for Elixir have their removal assisted by at least one refactoring also cataloged for this language.
- We showed that some refactoring operations cataloged for Elixir can be useful for addressing more than one code smell.
- On the other hand, we found that 12 of the 82 refactorings cataloged for Elixir are not associated with the removal of known code smells for this language.
- We identified five composite refactorings (*i.e.,* sequences of interrelated atomic[5] refactorings [178]) that are useful for removing code smells in Elixir.
- We have found evidence suggesting the existence of an uncatalogued smell for Elixir.
- We showed that the traditional refactorings, originally proposed to improve the quality of object-oriented systems [74], are also highly important for removing code smells in Elixir.

The results of this final working unit can guide developers—especially those new to Elixir—on how to systematically remove code smells and enhance the internal quality of their systems.

[5]Atomic refactorings are those not decomposable into other simpler code transformations [109].

1.2.4 Impact on the Elixir Developers' Community

In addition to the contributions from the three working units mentioned before, this Ph.D. thesis helped popularize discussions on software quality among developers working with Elixir. Some examples of this impact are listed as follows:

- Our GitHub repository created to catalog code smells for Elixir became popular among Elixir developers, receiving approximately 1.5k stars, thus ranking among the 60 most-starred Elixir GitHub-based projects.[6] Due to the interest sparked in the developer community for this content, part of this work was later incorporated into the official Elixir documentation through collaboration with the core team that maintains the language.[7] Unlike the taxonomy proposed in this thesis to classify code smells for Elixir (*i.e.,* Low-Level Concerns, Design-Related, and Traditional), the members of the Elixir Core Team chose to use an alternative taxonomy, composed of the categories Code-Related, Design-Related, Process-Related, and Metaprogramming-Related.

- The GitHub repository we used to catalog refactorings for Elixir has around 160 stars.[8] Although it has not become as popular as our code smells repository, some Elixir developers are already drawing inspiration from it to create tools capable of automatically applying some of the refactoring strategies cataloged in this thesis (*e.g.,* RefactorEx[9]).

- In mid-2022, we were interviewed on the podcast *Elixir em Foco*,[10] the main podcast of the Brazilian Elixir developers' community, where we were able to discuss the research conducted in this thesis. Additionally, our research was a topic of discussion on the *Thinking Elixir*[11] and *Elixir Mentor*[12] podcasts, two of the most well-known podcasts in the international Elixir developers' community.

- The author of this Ph.D. thesis gave a talk on Elixir-specific code smells and refactorings at Elixir Fortaleza Conf 2023.[13] This event was organized by members of the Brazilian Elixir developers' community. In addition to promoting the dissemination of these topics among developers, this opportunity allowed us to engage directly

[6] https://github.com/lucasvegi/Elixir-Code-Smells
[7] Official documentation: https://hexdocs.pm/elixir/what-anti-patterns.html
[8] https://github.com/lucasvegi/Elixir-Refactorings
[9] https://github.com/gp-pereira/refactorex
[10] Elixir em Foco: https://youtu.be/dp8zQUadDgQ
[11] https://podcast.thinkingelixir.com/93
[12] Elixir Mentor: https://youtu.be/BAchf-VSOhY
[13] Elixir Fortaleza Conf 2023 talk: https://youtu.be/klubcNmv4qI

with them to discuss issues related to the catalog of refactorings for this language, which was still a work in progress at the time.

- The research conducted in this thesis was also the topic of talks at ElixirConf 2023[14] and Code BEAM Europe 2023,[15] two of the main international events specifically geared towards developers working with Elixir. These talks were given by Elaine Watanabe, an experienced programmer specializing in Ruby and Elixir.

- Finally, José Valim, the creator of Elixir, was the keynote speaker at ElixirConf EU 2024. In his talk on design patterns for Elixir,[16] he also presented our work on code smells and refactorings, highlighting some of its key contributions and impacts on the developer community. Moreover, he described our research as a source of inspiration for discussions on other topics related to software quality in Elixir.

1.3 Publications

This thesis is based on the content of the following publications:

- **ICPC'22** Vegi, L. F. M. and Valente, M. T. Code smells in Elixir: early results from a grey literature review. In *30th International Conference on Program Comprehension (ICPC) - ERA track*, pages 580–584, 2022. doi: https://doi.org/10.1145/3524610.3527881. (**Chapter 3**).

- **EMSE'23** Vegi, L. F. M. and Valente, M. T. Understanding code smells in Elixir functional language. *Empirical Software Engineering*, 28(102):1–32, 2023. doi: https://doi.org/10.1007/s10664-023-10343-6. (**Chapter 3**).

- **ICSME'23** Vegi, L. F. M. and Valente, M. T. Towards a catalog of refactorings for Elixir. In *39th International Conference on Software Maintenance and Evolution (ICSME) - NIER track*, pages 358–362, 2023. doi: https://doi.org/10.1109/ICSME58846.2023.00045. (**Chapter 4**).

Furthermore, we also contributed to the following work during this Ph.D. research:

- **VEM'22** Nunes, H. G., Vegi, L. F. M., Cruz, V. P. G., and Figueiredo, E. Democracia em xeque: um estudo comparativo sobre detecção de code smells. In *10th*

[14] ElixirConf 2023 talk: `https://youtu.be/aSOY70vydp4`
[15] Code BEAM Europe 2023 talk: `https://youtu.be/6r5b574ttV8`
[16] ElixirConf EU 2024 keynote talk: `https://youtu.be/agkXUp0hCW8`

Workshop de Visualização, Evolução e Manutenção de Software (VEM), pages 11–15, 2022. doi: https://doi.org/10.5753/vem.2022.226562.

1.4 Outline of the Thesis

We organize this thesis as follows:

Chapter 2 covers background information to support this thesis. We provide an overview about the Elixir functional language, code smells, and refactoring. Additionally, we present a comprehensive review of relevant works directly related to the thesis. Finally, we emphasize the main distinctions between these works and our research.

Chapter 3 presents two studies conducted with the objective of characterizing code smells in Elixir. The first study involves prospecting, documenting, and cataloging code smells for Elixir. In the second study, we carry out a survey with developers to validate this catalog of code smells.

Chapter 4 introduces two other studies aimed at characterizing refactorings in Elixir. The first study focuses on identifying, documenting, and cataloging refactorings for Elixir. Each cataloged refactoring technique is accompanied by code examples and some specific side conditions tailored to the Elixir language. The second study involves conducting a survey with developers to validate this catalog of refactorings.

Chapter 5 reports a study on the interplay between code smells and refactorings in Elixir. Based on the relationships established by comparing the catalogs defined and validated in Chapters 3 and 4, we proposed practical guidelines for systematically removing code smells in this language. These guidelines suggest applying refactoring strategies in a specific order, one step at a time, to ensure a disciplined approach.

Chapter 6 summarizes the conclusions we leveraged throughout this thesis. It also outlines ideas we find interesting to investigate in the future.

Chapter 2

Background and Related Work

We begin this chapter by providing an overview of functional programming (Section 2.1) and Elixir language (Section 2.2). Then, in Section 2.3, we describe what code smells are and discuss the main types of investigations related to these sub-optimal code structures. Similarly, in Section 2.4, we delve into the concept of refactoring and explore its major fields of study. In Section 2.5, we present the works directly related to this thesis. Finally, we provide our final remarks in Section 2.6.

2.1 Functional Programming

The history of functional programming began in the 1930s when Church [48] introduced the *Lambda calculus*, which is used in computation to implement functions capable of accepting other functions as parameters and even returning functions after processing [6]. The first programming language to implement ideas influenced by the Lambda calculus was LISP, developed in 1958 and first presented in the early following decade by McCarthy [126].

While the ideas that formed the basis of the functional programming paradigm are older, it was only formalized a few years later by Backus [16]. In his work, Backus [16] presented various aspects of developing software through the combination of mathematical equations, coining the term *"functional"* to describe this paradigm and defining some of the guiding principles for its languages. Unlike the object-oriented paradigm, which stores states in objects and provides methods to modify those states, functional programming does not rely on this state-function dependency. This is due to the principle of **immutability**, which is one of the fundamentals of functional programming and determines that once data is defined, it should not change anymore [15]. When immutable data needs to be modified through an operation, functional programming languages create new data containing the transformed values, thus preserving the original data. According to Almeida [6], this feature benefits parallel and concurrent programming environments.

Since the creation of LISP, parallel processing has been a driving force behind software development using functional programming. Although historically functional languages have not been as popular in the industry as object-oriented languages, there has been a recent increase in interest in functional languages [26]. According to Swaine [179], with modern CPUs having an increasing number of cores and many systems requiring high availability for a large volume of concurrent users, modern functional languages such as Elixir[1] and Clojure[2] have emerged as viable options for such environments. These languages enable optimizing the utilization of computational resources, resulting in performance gains for the code [179]. In addition to immutability, functional programming is based on concepts such as pure functions, recursion, first-class functions, lazy evaluation, and pattern matching [15]. These concepts are described in the following paragraphs.

While in object-oriented languages we are more accustomed to assigning values to a variable as soon as it is defined, in functional languages a variable is not always computed at the moment of its definition. Through the concept of **lazy *(or nonstrict)* evaluation**, functional languages allow us to have variables that know how to compute the values that can be assigned to them, but only perform this computation the first time they are referenced in the code. Lazy evaluation can significantly optimize processing by avoiding unnecessary computations, especially when certain variables might never be used in specific code paths [15]. In some functional languages, such as Haskell,[3] lazy evaluation is the default behavior. In others, like Clojure, F#,[4] Scala,[5] and Elixir, we must explicitly specify when to use this feature [124].

In general, functions are considered **first-class citizens** in functional languages. This means they are treated as a data type and can therefore be assigned to variables [94]. Since functional languages have this characteristic, we can use them to define *higher-order functions*, which are those that take one or more functions as arguments or return a function as a result. According to Swaine [179], this powerful feature of function composition can be useful for increasing code reusability.

In functional languages, iterations are commonly performed via **recursive functions**, as these languages typically do not have classical iteration constructs like `while` and `do..while`, which are common in object-oriented languages that rely on mutable state [6]. However, functional languages also offer several higher-order functions that enable iteration while hiding the details of recursion, which can reduce the size and improve the understandability of the code, as will be demonstrated throughout this thesis. Examples of these functions include **`map`**, which applies a transformation to each element of a list, returning a new modified list; **`filter`**, which is used to select elements from a list

[1] `https://elixir-lang.org/`
[2] `https://clojure.org/`
[3] `https://www.haskell.org/`
[4] `https://fsharp.org/`
[5] `https://www.scala-lang.org/`

that satisfy a specific condition, creating a sublist based on logical criteria; and **`reduce`**, which condenses all elements of a list into a single value by accumulating the results of a user-defined operation [185].

Still regarding the behavior of functions in functional languages, whenever possible, they are defined as **pure functions**. A function is considered pure when, given the same set of inputs, it always returns the same result. Furthermore, a pure function should not cause *side effects*, meaning it should not have any interaction with the external world beyond its scope (*e.g.,* altering global variables or sending data over the network) or make modifications to the program's state (*e.g.,* saving to a file/database or displaying something on the screen). According to Backfield [15], although it is generally not possible to build a system composed solely of pure functions, the characteristics of these functions make them more predictable and easier to test.

Finally, **pattern matching** is a mechanism that allows a value to be compared with a specific pattern while simultaneously extracting information from it. This feature is used by functional languages to destructure complex data, facilitating access to specific parts of structures such as lists or tuples [160]. Based on the patterns found in the compared values, different actions can be executed by code that utilizes this feature. For this reason, pattern matching is also frequently used by functional languages as a control-flow mechanism instead of traditional conditional statements, such as `if..else` and `case` [6].

2.2 Elixir Language

Elixir is a modern functional programming language that performs well in parallel and distributed environments [94]. It was conceived in 2012 by José Valim, inspired by a mix of other languages like Erlang, Haskell, Clojure, and Ruby. Specifically, Elixir's syntax is Ruby-based, so it tends to be user-friendly. It uses immutable data, just like Haskell, making it well-fit 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 known to be robust, fault-tolerant, and powerful to run concurrent and distributed systems [6]. In addition, Elixir is a polymorphic and extensible language, as it has inherited features such as `protocols` and `macros` from Clojure. Elixir is primarily used to develop high-demand Web applications,[6]

[6] `https://www.phoenixframework.org/`

but it can also be used to develop machine learning systems,[7] embedded software,[8] code notebooks,[9] data science solutions,[10] and systems for many other purposes. As a result, although not yet mainstream, Elixir is becoming more popular in the industry, with over 300 companies worldwide using the language, including some well-known, such as Adobe, Cabify, Discord, Heroku, Motorola, PepsiCo, Pinterest, and Spotify.[11]

With Elixir, developers can create scalable and fault-tolerant concurrent systems more easily and with fewer computational resources. Instead of directly managing synchronization mechanisms such as semaphores, Elixir systems are based on units of concurrency known as BEAM processes [183]. 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 [94].

Elixir programs are organized by `modules`, which are groups of functions. Listing 2.1 shows an Elixir module (`Square`) composed of two functions—`area/1` and `perimeter/1`. In lines 10 and 11, these functions are called using *Elixir's interactive shell* (IEx),[12] which is an intelligent terminal that allows developers not only to run their code but also to test and access its documentation.

Listing 2.1: Example of code organization in Elixir

```
1  defmodule Square do
2    def area(side) do
3       side * side
4    end
5    def perimeter(side) do
6       side * 4
7    end
8  end
9  ...
10 iex(1)> Square.area(5)      #25
11 iex(2)> Square.perimeter(5) #20
```

Although Elixir does not support object creation, it allows modules to define `structs`, which are key-value pairs similar to objects. Listing 2.2 shows an Elixir module with a `struct` that represents a `Triangle`. This struct has three fields—`a`, `b`, and `c`—which are initialized to null values (`nil`) on line 2.

Listing 2.2: Examples of some of Elixir's features

```
1  defmodule Triangle do
2    defstruct [a: nil, b: nil, c: nil]
3
```

[7] https://github.com/elixir-nx/nx
[8] https://www.nerves-project.org/
[9] https://livebook.dev/
[10] https://github.com/elixir-explorer/explorer
[11] https://elixir-companies.com/en
[12] https://hexdocs.pm/iex/IEx.html

```
4    def scale_by(t, factor) do
5      %Triangle{a: t.a * factor, b: t.b * factor, c: t.c * factor}
6    end
7
8    def is_right_angled(t) do
9      [c1, c2, h] = Enum.sort([t.a, t.b, t.c])
10     Float.pow(c1, 2) + Float.pow(c2, 2)
11     |> equals(h * h)
12   end
13
14   defp equals(a, b) do
15     a == b
16   end
17 end
18 ...
19 iex(1)> tri = %Triangle{a: 4.0, b: 5.0, c: 3.0}  # struct creation
20 iex(2)> Triangle.scale_by(tri, 2)                # %Triangle{a: 8.0, b: 10.0, c: 6.0}
21 iex(3)> Triangle.is_right_angled(tri)            # true
22 iex(4)> Triangle.equals(2, 3)
23 ** (UndefinedFunctionError) Triangle.equals/2 is undefined or private
```

In Elixir, a struct has the same name as the module where it is defined. Besides the struct, the module `Triangle` also has three functions—`scale_by/2`, `is_right_angled/1` and `equals/2`. It is important to note that, unlike objects, structs are immutable data structures. For this reason, the `scale_by/2` function creates a new `Triangle`, instead of simply modifying the sides' values of an existing `Triangle` (line 5). In Elixir, `%T{...}` is analogous to `new T(...)` in an object-oriented language. In addition to being immutable, structs also differ from objects in that they do not support the *`this`* pointer, as in Java and C++, nor do they support instance variables. Therefore, the internal state of a struct is not available directly within the functions of the module where the struct is defined. If it is necessary to access the field values of a struct within these functions, they must receive a parameter of the struct's type, as in `scale_by/2` and `is_right_angled/1`.

The `is_right_angled/1` function needs to sort the sides of the triangle in ascending order before classifying it (line 9). This sorting is done by calling `Enum.sort/1`, provided by Elixir. It receives a list composed of the values of the three fields of the struct `Triangle` and returns a list with these values sorted. To facilitate direct access to these values from the returned list, they are extracted into three distinct variables—`c1`, `c2`, and `h`—using pattern matching, common in Elixir systems. A *pipe operator* (`|>`), another idiomatic feature of Elixir, is then used to make the nested calls of `Float.pow/2` in the `equals/2` call more natural (line 11). For example, in Elixir, `foo() |> bar(p)` is equivalent to `bar(foo(), p)`. For this reason, in the `equals/2` call, only one parameter is informed directly (line 11). Finally, `equals/2` is defined as a private function (line 14), and for that reason, it can only be called within the `Triangle` module.

As shown in Listing 2.3, 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.

Listing 2.3: Example of more 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]
```

Although Elixir has classical conditional constructs like `if..else` (lines 3 to 7), throughout this thesis, it will be shown that these constructs can be replaced by pattern matching, which, among other things, is often used as a control-flow mechanism in Elixir [94]. 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` 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).

2.3 Code Smells

Fowler and Beck [74] coined the terms *code (or bad) smells* to name sub-optimal code structures that can harm software maintenance and evolution [177, 212]. In addition to coining the terms, they cataloged 22 code smells, which are listed in Table 2.1. There are also other terms that are mostly synonyms of code smells, such as anti-patterns [40], code anomalies [136, 137] and bad practices [180].

Table 2.1: Traditional code smells

Code smells cataloged by Fowler & Beck [74]	
Duplicated Code	Switch Statements
Long Function	Lazy Class
Large Class	Alternative Classes with Different Interfaces
Long Parameter List	Incomplete Library Class
Feature Envy	Inappropriate Intimacy
Shotgun Surgery	Temporary Field
Divergent Change	Message Chains
Speculative Generality	Middle Man
Comments	Data Class
Data Clumps	Parallel Inheritance Hierarchies
Primitive Obsession	Refused Bequest

Many studies have been conducted in recent years aiming to detect, prevent, predict, remove, and understand the extent of the impacts caused by these structures on code quality. In addition to decreasing maintainability and hampering evolution, code smells can increase bug-proneness [115, 138]. According to Nagappan *et al.* [132], applying predictive strategies to anticipate the presence of sub-optimal structures in a codebase is one approach to mitigate the impact of code smells. To simplify the task of training and evaluating machine learning models used in the prediction of code smells, Santos *et al.* [158] compared which features and quality attributes are redundant or different among these models, and which of them contribute more to the predictions. Regarding code

smells removal, according to Liu *et al.* [116], the interrelation between code smells can be used to prioritize the removal of these code structures during refactoring activities. For example, removing instances of Duplicated Code can also promote the disappearance of instances of Long Function. Sobrinho and Maia [175] investigated the possibility of interrelation between three code smells in five open-source systems. More specifically, the authors sought to understand whether instances of the smells Large Class and Complex Class, with different intensities, can influence the presence of Duplicated Code. Some patterns of interrelationship between these three smells were found, such as classes with high complexities, regardless of their sizes, tend to have a higher prevalence of instances of Duplicated Code. According to the authors, patterns like this can be used to improve code smell detection techniques and tools.

Different techniques and tools for detecting code smells have already been proposed. They are mainly based on strategies that utilize software metrics, textual analysis, and AST analysis [66]. Chidamber and Kemerer [47] present commonly used metrics for detecting smells, such as Weighted Methods Per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), Coupling between Objects (CBO), Response For Class (RFC), and Lack of Cohesion of Methods (LCOM). Marinescu [122, 123] presents metrics-based approaches to detect the smells God Class and Data Class. Bavota *et al.* [20] use the McCabe cyclomatic complexity metric to detect the Complex Class smell. According to the authors, monitoring the evolution of metrics like this can also be used to prevent the introduction of code smells. Code smell detection is also performed in some studies using artificial intelligence techniques, such as in Fontana *et al.* [71], who conducted a study comparing 16 different machine learning algorithms to detect four types of code smells (*i.e.,* Data Class, Large Class, Feature Envy, and Long Method). Similarly, Cruz *et al.* [51] evaluated seven different machine learning algorithms on the task of detecting four types of code smells (*i.e.,* God Class, Long Method, Feature Envy, and Refused Parent Bequest), showing that with proper optimization, these algorithms can perform well. Regarding yet to the artificial intelligence-based detection techniques, White *et al.* [206] proposed a technique based on deep learning to detect the Duplicated Code smell. The aforementioned code smell detection strategies, along with others already studied, are summarized and categorized in some systematic literature reviews conducted to help researchers compare and understand existing methods and improve them [95, 171].

More than 80 tools designed for detecting code smells were published in the literature [66, 176]. According to Sobrinho *et al.* [176], DECOR [129] is the most frequently used tool for handling code smells in studies focused on detecting these sub-optimal structures. Detection tools do not always behave the same in a given context. For example, tools like PMD [70], JDeodorant [69], and JSpIRIT [202]—open-source plugins for the Eclipse IDE [61] that can statically analyze Java code—may detect different instances

of the God Class smell [151] when analyzing the same code [148]. This discrepancy can occur due to the different metrics and thresholds employed in their detection strategies. There are also solutions that detect smells by intersecting the results from multiple tools. In these approaches, if at least 50% of the tools identify a smell, the element is considered problematic, similar to a voting system [51, 87].

According to Fontana *et al.* [72], the impacts on software quality caused by code smells are not homogeneous and may differ among domains. Furthermore, developers' perception of code smells can also vary across different software contexts [181]. For this reason, code smells for specific contexts like Android [83], iOS [82], JavaScript [63, 67, 155], and others have also been studied. Although there is vast research on code smells for specific contexts [27, 63, 67, 82, 83, 133, 146, 155, 165], it is mostly focused on the object-oriented paradigm. Sobrinho *et al.* [176] carried out a systematic review of the literature on articles about code smells published between 1990 and 2017. Exactly 104 code smells were cataloged in this period, however, none of them refer to the functional paradigm or to languages that follow this paradigm. Indeed, a technical report by Cowie [50] focusing on Haskell, and the works published by Li *et al.* [108, 113] centered on Erlang, are the only ones we found that address specific smells of functional languages.

The systematic literature review conducted by Sobrinho *et al.* [176] also highlighted several other research opportunities involving code smells, such as: studying the interaction between less widely known code smells and the more popular ones in the literature to understand their combined impacts; investigating the impact of code smells in specific contexts; identifying novel code anomalies in these specific contexts; and documenting new catalogs of code smells specific of a context, as we do in this thesis for the Elixir functional language (Chapter 3).

2.4 Refactoring

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 [74]. The term was first proposed by Opdyke [141] and popularized by Fowler in his well-known catalog containing 72 refactorings for object-oriented code [74]. According to Fowler [74], Ward Cunningham and Kent Beck were among the first people to understand the importance of refactoring. They applied these techniques within the Smalltalk ecosystem starting in the 1980s, thereby influencing the notion of refactoring to become a significant element of Smalltalk culture. However, the concept of *program transformations* has been discussed since the 1970s by Burstall and Darlington [42]. This

term is equivalent to refactoring and commonly used in the context of functional languages [85, 144].

According to Thompson [184], 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 performed refactoring by developers [79]. 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 widely 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 [32, 178], assessing their impact on software quality [4, 7, 24, 25, 149, 205], developing tools for detecting and/or automatically performing refactorings [36, 112, 119, 153, 156, 166], comprehending the reasons why practitioners perform refactoring [1, 169], and also investigate how and when refactorings are performed [131, 139, 209].

According to Abid *et al.* [1], the primary motivation for developers to refactor their code is the pursuit of improving the internal quality of systems (41.6%), followed by improving external quality (22.7%), and enhancing performance (16%). With the same goal of understanding the motivations behind refactoring operations applied by developers, Silva *et al.* [169] monitored 748 Java projects maintained on GitHub to detect refactoring activities and then requested developers from these projects to explain the reasons for the performed refactorings. The authors identified 12 refactoring strategies applied in these projects, all cataloged by Fowler [74]. In total, the authors cataloged 44 different motivations for these refactorings, most of which were related to changes in requirements and business rules, and less related to quality improvements, such as removing code smells.

Almogahed *et al.* [7] investigated how specific refactoring techniques can impact software reusability. The authors selected five traditional refactoring strategies cataloged by Fowler [74]—EXTRACT INTERFACE, ENCAPSULATE FIELD, EXTRACT CLASS, INLINE CLASS, and INLINE METHOD—and applied them about 300 times to the open-source code of the well-known text editor JEDIT.[13] The impacts of this experiment were quantified using software metrics to measure reusability, and it was shown that while some refactoring strategies can significantly improve reusability (*i.e.,* EXTRACT INTERFACE, ENCAPSULATE FIELD, and EXTRACT CLASS), others can harm this software quality attribute—INLINE CLASS and INLINE METHOD.

Additionally, the correlation between refactoring activities and the introduction of bugs has been investigated by Weißgerber and Diehl [205]. This study compared the refactoring history of three open-source systems (JUNIT, JEDIT, and ARGOUML)

[13] https://sourceforge.net/projects/jedit/

with their respective bug reports. For most historical phases, an increase in refactoring activities did not lead to a higher bug rate in these systems. In a similar study, Ratzinger *et al.* [149] concluded that an increase in refactoring activity tends to be followed by a decrease in software defects.

In addition to documenting their well-known catalogs of code smells and refactorings, Fowler and Beck [74] establish a relationship between these catalogs, characterizing code smells as opportunities for refactoring. For each of the 22 traditional code smells, the authors suggest refactoring techniques to remove them, which can be either atomic (*i.e.,* not decomposable into simpler refactorings [109]) or composed of a set of these atomic ones. Some studies investigate issues related to composite refactorings, which are refactorings of larger granularity, characterized by being sequences of atomic refactorings as those proposed by Fowler [74]. According to Souza *et al.* [178], all refactorings in a composite can either be of the same type (*e.g.,* multiple PULL UP METHOD) or not (*e.g.,* multiple MOVE operations eventually followed by a RENAME, or a sequence of EXTRACT METHOD operations followed by MOVE METHOD operations). These authors refer to this characteristic as *composite uniformity*. Bibiano *et al.* [24] identified the most common types of incomplete composite refactorings and their impact on software quality attributes such as coupling and cohesion. Refactorings are considered incomplete when they fail to remove a code smell completely. The authors analyzed 353 incomplete refactorings related to the removal of the smells FEATURE ENVY and GOD CLASS in five different systems, concluding that the main reason for the failure to remove these smells are incomplete composite refactorings with at least one EXTRACT METHOD applied without MOVE METHODS. On the other hand, the authors concluded that most incomplete composite refactorings at least do not worsen the quality attributes of a code. According to Cedrim *et al.* [43], incomplete refactorings can be very frequent. In an analysis of 16,566 refactorings along the version histories of 23 projects, the authors identified that although approximately 79% of the refactorings modified code that contained smells, only 9.7% of these refactorings completely eliminated these smells.

Regarding yet to the composite refactorings, Brito *et al.* [32] proposed a catalog containing eight of these strategies. The authors detected these cataloged composite refactorings through a set of scripts integrated into REFDIFF [33, 168, 170], which is a tool that automates the identification of refactoring operations performed in the version history of systems developed in multiple languages such as Java, JavaScript, Go, and C. Other similar tools for detecting refactorings, such as REFACTORINGMINER [13, 191], REFACTORING CRAWLER [58], REF-FINDER [97], REFACTORINSIGHT [103], and RMINER [190], have also been proposed in the literature. Oliveira *et al.* [140] sought to better understand the detection capability of the REFDIFF and REFACTORINGMINER tools. To do so, the authors conducted a survey with 53 developers who work on popular open-source projects implemented in Java and asked them to identify refactorings applied to a codebase. Many

of the manual detections performed by developers were not detected by the studied tools, thus demonstrating that there are still research opportunities related to improving the automatic detection capability of refactorings.

Concerning tools and libraries that support automatic refactoring, there are options available for various programming languages. For example, for C# and .NET, there is ReSharper [90]. Smalltalk developers can utilize the Refactoring Browser [152]. Java developers can benefit from tools such as IntelliJ IDEA [89], Eclipse IDE [61], NetBeans [134], and JastAdd Refactoring Tools (JRRT) [159]. More specifically, mobile developers who use Java for Android development or Objective-C and Swift to develop iOS applications can utilize a tool called Piranha [147]. The refactoring library Piuma [166] can be used in code implemented in Scala. Erlang programmers have access to tools like Wrangler [109, 112, 113], Tidier [156], and RefactorErl [119]. Haskell developers, on the other hand, benefit from a range of tools, including HaRe [36, 39, 111], Programming Assistant for Transforming Haskell (PATH) [192], Ulm Transformation System (Ultra) [81], HERMIT [64, 65, 161], and the Haskell Equational Reasoning Assistant (HERA) [78]. Meanwhile, OCaml developers can leverage the Rotor tool [153, 154].

These tools that support automatic refactoring have a limited number of implemented refactoring strategies. Before executing any of these strategies to transform a code, these tools check for certain conditions to ensure that the transformed code will preserve the same behavior as the original one. Soares *et al.* [174] proposed a technique based on differential testing [127] to quantify the automatic refactorings rejected by the Eclipse IDE, NetBeans, and JRRT that are related to overly strong preconditions, *i.e.,* unnecessary conditions to preserve code behavior after refactoring. The authors identified 24 types of overly strong conditions that limit the applicability of Eclipse IDE and JRRT to perform automatic refactorings. Similarly, Mongiovi *et al.* [130] proposed a technique to detect overly strong preconditions in refactoring implementations by disabling preconditions. The authors evaluated this technique by quantifying automatic refactorings rejected by the Eclipse IDE and JRRT, finding 14 overly strong preconditions in Eclipse IDE and four in JRRT. In addition, the authors compared this technique based on disabling preconditions with the technique proposed by Soares *et al.* [174] that makes use of differential testing, concluding that these techniques are complementary. Although they have common detections, there are also detections found exclusively by each of the techniques, justifying their joint use by developers.

Since refactorings are code transformations that should not alter the observable behavior of a program, some studies have specifically focused on validating behavior preservation (*i.e.,* correctness) in refactored code. Bereczky *et al.* [22, 23] introduced a machine-checked formalization of Core Erlang, a subset of the Erlang language, to establish concepts of program equivalence useful for formally proving that specific refactoring

strategies preserve program behavior in Erlang. This formalization includes rigorous mathematical definitions of syntax and semantics for Core Erlang. The authors utilized this to prove the correctness of simple refactorings in Erlang.

In contrast, Seres *et al.* [163] focused on verifying behavior preservation in refactored Erlang code through extensive testing. They developed a tool called EQUIVCHECKER, which can be used as an extension of VSCODE IDE or integrated into CI pipelines. This tool detects changes in the code and compares the old and new versions of all affected functions. To determine whether the behavior has been preserved, the tool uses *property-based testing*, generating a large number of random test cases that automatically validate the correctness of refactorings applied to Erlang code.

Refactoring activities are frequently performed [205, 209]. However, despite the existence of several tools to assist in refactoring activities, according to Murphy-Hill *et al.* [131], approximately 90% of refactorings are performed manually by developers. In their research involving Java projects maintained on GitHub, Silva *et al.* [169] also observed that manual refactorings are more prevalent, representing 55% of the refactoring operations performed. These results contrast with those presented by Oliveira *et al.* [139], where the authors conducted a survey with 107 developers and identified that approximately 75% of them use IDEs to apply refactorings.

An increasing number of studies are investigating the use of artificial intelligence strategies in the software refactoring process. Considering that state-of-the-art tools aimed at identifying refactoring opportunities (*e.g.,* code smells) still present a high number of false positives [92], in real-world scenarios, the identification of these opportunities still heavily relies on the expertise and intuition of developers. Aiming to understand how supervised machine learning (ML) algorithms can support developers in making faster and more informed decisions regarding what to refactor, Aniche *et al.* [9] trained models using six different ML algorithms with a database containing over two million refactorings performed in nearly 11.2k open-source projects. The authors believed that by training these models with refactored classes and methods from real-world projects, they could provide more reliable refactoring recommendations to developers. Indeed, this study produced models capable of identifying opportunities for applying 20 different refactoring strategies with an accuracy often higher than 90%.

Al-Fraihat *et al.* [5] proposed a technique based on four ML algorithms to detect whether refactorings occurred in a commit and determine which types of them were performed. The authors trained their models using a dataset of 573 commits from three Python projects to detect and classify refactorings. Their technique recognizes and categorizes nine different traditional refactorings as cataloged by Fowler [74]. Among the algorithms compared, XGBoost achieved the best performance, with an accuracy of 100%.

Since *Large Language Models* (LLMs), such as ChatGPT, have become widely popular, they have been applied to a variety of software engineering tasks, including

refactoring. AlOmar *et al.* [8] conducted an exploratory study aimed at understanding how developers interact with ChatGPT during refactoring activities. Specifically, the authors sought to understand how developers describe their refactoring needs to ChatGPT, how they initiate conversations with the LLM when seeking help, and what key quality attributes are considered by ChatGPT in its responses. To achieve this, the authors mined a dataset composed of 176 code files, 470 commits, and 69 issues from open-source GitHub projects. All of these artifacts are associated with nearly 18k developer-ChatGPT conversations (*i.e.,* prompts and responses) concerning their refactorings.

The aforementioned study found that, although it is not uncommon for developers to use generic terms to describe their refactoring needs in prompts (*e.g.,* cleanup and improve code quality), they more frequently use specific operation names following Fowler's conventions [74], indicating that developers are familiar with the cataloged refactorings and use them in their communications. Additionally, the results showed that while 41.9% of the prompts contain a code snippet to be changed along with a detailed textual description of how developers want it refactored, approximately 35% of the prompts overestimate the model's capabilities, such as by providing only a textual description of the code without the actual code snippet. Finally, the study revealed that ChatGPT considers a wide range of internal and external quality attributes in its responses.

According to Bordignon and Silva [26], an increasing number of developers are using functional languages in the industry. Conversely, the systematic literature review conducted by Abid *et al.* [1] 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 taking advantage of in this thesis (Chapter 4).

2.5 Related Work

In this section, we discuss work related to this thesis. Since there is a scarcity of studies conducted on code smells and refactorings for functional languages, and to the best of our knowledge, no studies in these fields specifically tailored for Elixir, in the following subsections, we present the works we identified as relevant to this thesis in a more general way. First, we discuss studies about context-specific code smells, highlighting topics related to our research and the main differences to our catalog of Elixir-specific smells (Subsection 2.5.1). Next, we present studies about language-specific refactoring strategies (Subsection 2.5.2).

2.5.1 Context-specific Code Smells and Developers' Perceptions

As code smells are context-sensitive, other studies were carried out to catalog code smells in specific domains. Reimann *et al.* [150] proposed a catalog with 30 Android-specific code smells, extracted from the grey literature. Hecht *et al.* [83] selected four of these Android-specific smells and tried to identify them in real projects using a detection technique based on code metrics. Also on code smells specific to mobile platforms, Habchi *et al.* [82] cataloged six iOS-specific smells through a grey literature review and later validated them by analyzing their prevalence in 279 iOS repositories on GitHub.

Some studies have cataloged code smells specific to Web platforms. Fard and Masbah [63] proposed a set of 13 code smells for JavaScript, seven out of them are adapted from traditional smells, and six are specific to the language, extracted from the grey literature. They also analyzed 11 Web applications from different domains, seeking to detect these smells through a strategy based on code metrics. Closely related, Ferreira and Valente [67] proposed a catalog with 12 React-related code smells identified by a grey literature review and by interviewing developers. Afterward, they implemented a tool to detect these smells in the top-10 most popular GitHub projects that use React. Similarly, Saboury *et al.* [155] cataloged 12 code smells for JavaScript and validated them in five popular Web applications on GitHub. Still on specific code smells for Web applications, Punt *et al.* [146] cataloged 33 smells for CSS by reviewing the scientific and grey literature. They were grouped into seven different categories and later detected in 41 real Web applications available on GitHub.

Mashiach *et al.* [125] selected 35 traditional code smells among those cataloged by Fowler and Beck [74] and Brown *et al.* [40], and mined them in 44 real open-source projects implemented in C++ to validate a tool for detecting these sub-optimal structures in this language. There are other works that have cataloged code smells in even more specific contexts, such as the configuration management language Puppet [165], the command language Bash [59], quantum computing programs [46], machine learning systems [77, 213], and specific smells for video game development [3, 27, 28, 133]. In general, as in our work (Chapter 3), these papers also use grey literature or investigate real projects through repository mining techniques to catalog code smells from specific contexts.

Just like our work (Chapter 3), which conducted a survey to reveal the developers' view of which smells are more prevalent and which have the most negative impact on the maintenance of Elixir systems, other studies also sought to map the perception that developers have about code smells. Nardone *et al.* [133] cataloged 28 specific code smells for video game development and later validated these smells by applying questionnaires to 76 professionals in the area. The perception of these professionals regarding code smells served not only to validate the catalog but also to improve it with suggestions

for preventing and refactoring these smells. Similarly, Chen *et al.* [46] cataloged eight quantum-computing-specific smells extracted from grey literature and validated them through a survey with 35 quantum-computing developers. Subsequently, the authors analyzed the prevalence of these quantum-computing-specific smells by mining them in 15 open-source quantum programs.

Other studies sought to assess developers' perception of traditional code smells [12, 121, 143, 210, 211]. Taibi *et al.* [181] conducted surveys with experienced developers to understand their views on the negative impacts that smells can bring to software evolution and maintenance. Although most developers considered smells as harmful when they were analyzing their descriptions, few had the same perception when they analyzed chunks of code containing the same smells. That is, developers tend to see code smells more harmful in theory than in practice. In our study (Chapter 3), in order to express their perceptions, developers had access simultaneously to the smells descriptions and to code examples containing these sub-optimal structures.

About code smells specific to functional languages, Cowie [50] describe a tool for detecting eight code smells in Haskell. The author defined these smells based on code implemented by first-year undergraduate Computer Science students at the University of Kent. The most frequent structures that were inefficient or did not follow Haskell coding conventions were marked as code smells. Most of these smells affect small code structures, thus having a granularity equivalent to our Low-Level Concerns smells (Chapter 3).

2.5.2 Language-specific Refactorings

To the best of our knowledge, our study (Chapter 4) is the first one that catalogs refactorings for Elixir. Other research, however, provides transformation strategies and tools for other functional languages like Erlang [14, 31, 60, 84, 98, 100, 109, 112, 113, 114, 117, 118, 119, 156, 157, 182, 187], Haskell [36, 39, 62, 64, 65, 78, 81, 104, 105, 111, 161, 184, 186, 192, 203], ML [21, 93], OCaml [153, 154], Scala [166], and Racket [44]. In addition, some studies have investigated refactorings in Ruby, which is a non-functional language that influenced the creation of Elixir's syntax [11, 49].

Li *et al.* [109, 112, 113] present an automatic refactoring tool for Erlang called Wrangler. To illustrate the use of this tool, some refactoring strategies for this language are presented. Similarly, Sagonas and Avgerinos [156] and Lövei *et al.* [119] respectively propose the tools Tidier and RefactorErl, accompanied by refactoring strategies for Erlang. Regarding studies on refactorings for Haskell, Brown *et al.* [36, 39] and Li *et al.* [111] briefly described the refactorings implemented in the HaRe tool, a Haskell

refactorer. These refactoring strategies for Haskell and Erlang were compared by Li and Thompson [106], 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 [29, 30, 35, 38, 88, 101, 102, 189] and Haskell [19, 37]. Additionally, Chechina *et al.* [45] presented a systematic and tool-based approach for refactoring distributed Erlang applications. This approach uses some traditional refactorings that also belong to our catalog (Chapter 4), such as Rename an Identifier, Extract Function, and Moving a Definition. Furthermore, Li and Thompson [110] 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 (Chapter 4).

Similarly to what we did in our study presented in Chapter 5, the use of refactorings to remove code smells has been explored in studies involving functional languages other than Elixir. Li *et al.* [113] present a list of refactoring strategies capable of removing seven process-related smells specific to Erlang. The authors also discuss the challenges of automatically performing these refactorings due to Erlang's dynamic nature, the implicit nature of processes, and the communication structure between them in this language. Additionally, Li and Thompson [108] present other refactoring strategies for Erlang that can be semi-automatically performed by the Wrangler tool to remove four Erlang-specific smells related to the modularity of systems developed in this language. These code smells are similar to our Design-related smells (Chapter 3). Other studies focus exclusively on refactoring the traditional code smell Duplicated Code in Erlang [73, 107, 162, 188] and Haskell [34].

The aforementioned studies were retrieved by our systematic literature review (SLR), and some refactoring strategies outlined in them were adapted to our catalog of refactorings for Elixir (Chapter 4). 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.* [1] 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 [4] 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 [171] 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.* [2] 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 (Chapter 4).

2.6 Final Remarks

This chapter presents an overview and work related to the central themes addressed in this thesis. We begin by presenting a background of the functional programming paradigm, the main practical applications of some functional languages, and the principles that characterize languages in this paradigm (Section 2.1). In Section 2.2, we introduce the Elixir functional language, highlighting some of its key features through code examples. In Sections 2.3 and 2.4, we present overviews of code smells and refactorings, respectively. Besides defining these concepts, we also highlight the main types of investigations related to these fields of study. Finally, we concluded by addressing and discussing in Section 2.5 the studies closely related to this thesis.

While there are many investigations on code smells and refactoring strategies in the literature, to the best of our knowledge, only a scarce amount of them directly address functional languages, and none specifically target Elixir. Since Elixir is a language applicable to many programming purposes and has been becoming increasingly popular in the industry, we aim to fill this gap by generating knowledge that allows for the improvement of software quality implemented with this language.

Chapter 3

Code Smells in Elixir

In this chapter, **we propose a catalog composed of 35 code smells**, 23 of them are new and specific to Elixir, and 12 of them are traditional code smells, as cataloged by Fowler and Beck [74], which also affect Elixir systems. To accomplish this, we conducted two studies, with the first one focused on prospecting and documenting code smells for Elixir, and the second one aimed at validating them with developers.

This chapter is organized as follows. In Section 3.1, we present our catalog of code smells for Elixir. Also, we detail our mixed methodology, based on a grey literature review, the interaction with the Elixir developer community, and on the mining GitHub repositories to prospect and document code smells for this language. Additionally, this section presents the threats to validity associated with prospecting and documenting our code smells. In Section 3.2, we go into depth on the survey we conducted with developers to validate our catalog of Elixir smells. This section presents the survey findings, discusses the threats to validity, and outlines the methods employed in designing the questionnaires, reaching out to respondents, and analyzing the answers. In Section 3.3, we discuss the implications of our results. Finally, we conclude this chapter in Section 3.4.

3.1 Catalog of Code Smell for Elixir

In this section we present our first study, which focused on prospecting and documenting code smells in Elixir, thus proposing a catalog. This study was supported by qualitative data, letting the findings emerge from observations, providing a better understanding of the problem [208]. In this context, previous research cataloged code smells for specific-contexts [63, 82, 83, 146, 155, 165], but none consider the Elixir functional language. Therefore, in this study, we explore the code smells commonly discussed in the Elixir context. Particularly, we decided to use a mixed methodology to answer two key research questions:

RQ1. Do Elixir developers discuss traditional code smells? In this RQ, we seek

to understand whether the 22 code smells proposed in the nineties for object-oriented languages by Fowler and Beck [74] are important in the Elixir context.

RQ2. Do Elixir developers discuss other smells? Next, we investigate discussions about design and code problems specific to Elixir systems, thus referring to them as Elixir-specific smells.

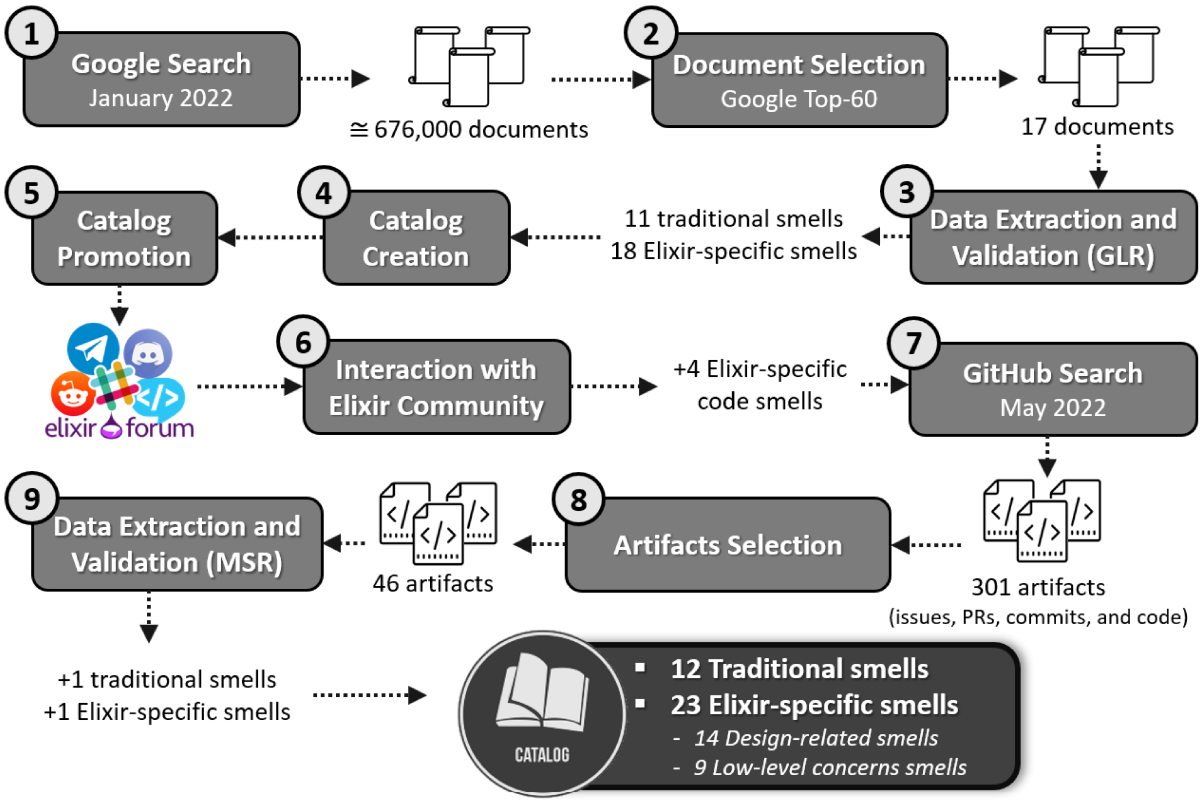
We dedicate Section 3.1.1 to present the mixed methodology applied to proposing a catalog of code smells for Elixir. In Section 3.1.2, we present the 12 traditional smells discussed by developers in the Elixir context. Next, in Section 3.1.3, we present a catalog composed by 23 Elixir-specific smells that emerged from our studies. We also classify these novel smells into two groups (Design-Related and Low-Level Concerns smells). Finally, Section 3.1.4 discusses threats to validity.

3.1.1 Study Design

Since Elixir is a new programming language, we have few scientific articles investigating software engineering and quality aspects of Elixir systems. For this reason, to prospect, document, and catalog code smells in Elixir, we use a mixed methodological approach, based on a grey literature review (GLR), interactions with Elixir community, and mining software repositories (MSR). Figure 3.1 summarizes the steps we followed to propose the catalog of code smells for Elixir. We also detail all these steps in the following paragraphs.

1) Google Search: In this step, we begin by reviewing the grey literature—composed of blogs, forums, videos, podcasts, etc.—in order to find discussions that characterize code smells in Elixir. Considering that few scientific articles investigate software engineering and quality aspects of Elixir systems, the grey literature is an interesting source of information for our goals [96, 214]. According to Garousi *et al.* [76], when defining the keywords for a Google search in a grey literature review, it is important to perform preliminary experiments to calibrate the queries, in order to combine synonyms or to exclude specific terms that might affect the results. Therefore, we started with the query presented in Listing 3.1.

Figure 3.1: Overview of methods for cataloging code smells in Elixir



Listing 3.1: Search query before calibration

```
1  ("Elixir" OR "phoenix framework") AND
2  ("code smell" OR "bad smell" OR "anti-pattern" OR "antipattern")
```

During the calibration, to improve the query results, the term *"phoenix framework"* was removed. This term was previously used because Phoenix has a large community, being the most used framework for developing Web apps in Elixir. However, its presence was polluting the results, returning many links related to an old Microsoft's technology[1] with the same name and that is not related to our research. Listing 3.2 presents the search query after this calibration.

Listing 3.2: Search query after calibration

```
1  ("Elixir") AND
2  ("code smell" OR "code smells" OR "bad smell" OR "bad smells" OR "anti-pattern" OR
       "anti-patterns" OR "antipattern" OR "antipatterns" OR "bad-practice" OR  "bad-
       practices" OR "bad practice" OR "bad practices")
```

This query contains code smells synonyms, including a novel one (*"bad practice"*), both in the singular and in the plural. This was done to mitigate the risk that some desired discussions would be ignored. The final search was carried out in January 2022.

[1] https://web.archive.org/web/20071119175240/http://connect.microsoft.com/Phoenix

2) Document Selection: As approximately 676,000 documents were retrieved, it would be impractical to analyze all of them. This is a recurring problem in grey literature reviews, for this reason, according to Garousi *et al.* [76], it is necessary to limit the number of documents to be analyzed. As Google is based on the PageRank algorithm, which returns results in descending order of importance [142], it is natural that the further a document is from the beginning of the ranking of results, the greater the chances that it is out of context. Based on this premise, we established 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.

We analyzed the documents. In particular, our selection criteria were based on an adaptation of the Quality Assessment Checklist proposed by Garousi *et al.* [76]. For a document to be considered valid, first it should be related to the context of our research questions (RQ1 and RQ2). Second, it should meet at least one of the following Authority of the Producer criteria of Garousi's checklist: (a) is the publishing of a company that works with Elixir?[2] (b) is the author associated with a company that works with Elixir? (c) has the author published other works in the field? (d) does the author have expertise in the area?

After inspecting the top-60 documents returned by Google, *i.e.,* the first six pages of results, our stopping criterion was met, and 17 valid documents were selected, which we refer to as G1 to G17. All 43 documents that were discarded did not meet our first selection criterion, meaning they were not related to our RQs.

3) Data Extraction and Validation (GLR): The documents selected in the previous step were analyzed in detail by the author of this work, in order to identify discussions on traditional or novel code smells. The discussions were then validated by the advisor of this thesis. We discarded only two out of 17 documents—G4 and G15—due to a lack of agreement between the author and the advisor.

An example of disagreement can be seen in G15. In that document, posted on Reddit, a developer started a discussion that could indicate a novel Elixir-specific code smell related to the `with` statement, an Elixir control-flow structure:

> *"I think the "with" statement is an anti-pattern. Saying that this follows the let it crash principle is misleading. [...] having this construction makes it easier to abstract patterns in the wrong way. For example, if you get to the point where you have a lot of "with" clauses, you write code for handling all these errors, or you just swallow them and continue. It feels a lot like try-catch but for pattern matching results. [...] You can end up with a lot of complex synchronous code as a result."*

At Reddit, this post generated dissenting views, with some developers agreeing. However, other developers disagreed, as they think this is just a personal implementation

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

preference. This same type of disagreement occurred between the author and the advisor of this work and for this reason, we decided to discard the document and the discussed smell.

In total, 29 code smells emerged from our analysis in this step.

4) Catalog Creation: We analyzed the 29 code smells found in the previous step and classified 11 of them as traditional smells, as proposed by Fowler and Beck [74]. The other 18 smells were classified as Elixir-specific smells. The latter were also categorized into two different groups, according to the granularity of the structures they affect. The Design-Related group involves 10 smells related to code organization issues and therefore affects larger chunks of code. These smells can harm code readability or maintainability, for example. On the other hand, the Low-Level Concerns group is composed of eight smells that are more simple and that affect small code structures.

Table 3.1 presents the topics used to document each code smell of our catalog, which is also available in a GitHub public repository [195].

Table 3.1: Sections of the catalog of Elixir-specific code smells

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

5) Catalog Promotion: We also promoted the catalog in the main Elixir communication channels listed in the language official website. These channels are presented in Table 3.2.

Table 3.2: Elixir communication channels used to promote the initial catalog

Channel	**URL**
Telegram	`https://t.me/elixir_world`
Discord	`https://discord.gg/elixir`
Slack	`https://elixir-slackin.herokuapp.com`
DevTalk	`https://devtalk.com/elixir`
Reddit	`https://www.reddit.com/r/elixir/`
Elixir Forum	`https://elixirforum.com/`

We posted a message on these channels, inviting Elixir developers to browse our catalog on GitHub and to open Issues and Pull Requests suggesting improvements,

new code smells, and refactorings. These messages were posted between March and May 2022. During this period, influential people from the Elixir community, such as the creator of the language, became aware of our research and helped us to promote the catalog spontaneously. As a result, our repository became popular on GitHub, receiving around 950 stars over that period, thus ranking among the 100 most-starred Elixir GitHub-based projects.

6) Interaction with Elixir Community: In total, 25 documents—13 ISSUES and 12 PULL REQUESTS—were created by the community in the GitHub repository, which we refer to as E1 to E25, and 83 comments were made on them. All these documents underwent collective validation by the community, as well as by the author and the advisor of this study, seeking to select only those related to internal quality problems and not personal implementation preferences, for example.

As shown in Table 3.3, 20 documents were accepted by the author and the advisor of this study, resulting in 27 improvements in the catalog. Therefore, some documents have more than one contribution. In this table, contributions classified as OTHERS involve, for example, typo corrections, and improvements to formatting and organization. Two contributions of this type stand out. In E22 and E23, a developer suggested an adaptation of the catalog to the interactive format provided by Livebook,[3] a tool to write articles with adaptable and runnable code. With this format, each reader can create their own catalog instance, where code smell examples can be modified and executed interactively, thus enriching the learning experience.

Another relevant improvements were the addition of four Elixir-specific code smells—E3, E4, E6, and E7—expanding the catalog to a total of 22 specific code smells, including 14 DESIGN-RELATED and eight LOW-LEVEL CONCERNS smells. In addition to these four new Elixir-specific smells, three others were suggested in documents E9, E11, and E13, but were not validated by the community, nor by the author and the advisor of this study, as they were considered personal implementation preferences rather than internal code quality issues. An example of this can be seen in E11. In this document, a community member suggested a smell related to the use of the *pipe operator*, receiving the following comment by another member and which also represents the conclusion of the author and the advisor of this work:

> *"I don't [think] this is a smell either. I also think it mostly boils down to syntax preference and not really related to code design or code quality."*

Although in E2 and E7, developers have discussed two traditional code smells, one of them being present in both documents, they were not new to our research, as they had already been cataloged in our grey literature review.

[3] `https://livebook.dev/`

Table 3.3: Contributions by the Elixir community

Type	**Issue**	**Pull Request**	**Total**
	Sub.(*Acc.*)	*Sub.*(*Acc.*)	*Sub.*(*Acc.*)
Refactorings	3(2)	1(1)	4(3)
Smell Rename	1(1)	1(1)	2(2)
Bug Fixes	1(1)	2(2)	3(3)
Elixir-Specific Smell	7(4)	0(0)	7(4)
Traditional Code Smell	2(3)	0(0)	2(3)
Description Improvements	5(4)	0(0)	5(4)
Others	0(0)	8(8)	8(8)
Total	19(15)	12(12)	31(27)

Sub.: Submitted. / *Acc.*: Accepted.

7) GitHub Search: Aiming to expand the code smells catalog by mixing prospecting methodologies, at this step we mined software repositories (MSR) on GitHub. According to Dabic *et al.* [56], there are two main steps for mining repositories. First, researchers must define a repository selection criterion to filter only those related to their research questions. After that, researchers must execute a search query to retrieve contextualized data from the repositories.

To find artifacts—Issues, Pull Requests, Commits, and Files—that contain references to code smells, we used the same query from Step 1 (Listing 3.2). To execute this query, we use the standard GitHub search service. All searches were performed in May 2022, and we also used GitHub search qualifiers to filter the query keywords only in repositories where Elixir is the main language or in files with Elixir's extensions (*.ex* and *.exs*).

8) Artifacts Selection: A total of 301 artifacts were retrieved in the previous step. All were inspected by the thesis' author and his advisor seeking an agreement to select only those that meet a set of criteria.

As in Step 2, in order for an artifact to be selected, it must first be related to the context of our research questions (RQ1 and RQ2). Second, the artifact cannot have been previously retrieved in our grey literature review or be part of our GitHub repository created in Step 4 [195]. Third, we adapted Garousi's Authority of the Producer criteria [76] for the context of this MSR step. Therefore, in order to be selected, an artifact must meet at least one of the following criteria: (a) was the artifact published in the repository of a company that works with Elixir? (b) is the author a developer associated with a company that works with Elixir? (c) does the author have code repositories implemented in Elixir? Finally, to be selected, an artifact cannot be duplicated in relation to others already selected. These duplications can occur due to forked repositories on GitHub.

At the end of this step, 46 artifacts were selected and identified with keys ranging

from M1 to M46. Table 3.4 provides an overview of the retrieved and selected artifacts. Among the 255 discarded artifacts, 180 were not related to the context of our RQs, 65 were fork's duplications, and 10 were from our repository [195] or just direct links to it.

Table 3.4: Overview of artifacts selection

Artifact	**Retrieved**	**Selected**
Source Code Files	57	5
Commits	22	2
Issues	99	17
Pull Requests	123	22
Total	301	46

To ensure that these 46 artifacts meet our third selection criterion, we checked the author' profiles on GitHub, more specifically in the Bio field or in repositories they collaborate. After that, we concluded that all these 46 artifacts were also valid according to our criterion. For example, we selected artifacts created by Elixir-based companies, such as Finbits;[4] and from very popular repositories, such as Elixir Language[5] or Phoenix framework.[6]

9) Data Extraction and Validation (MSR): After selecting 46 artifacts in the previous step, the author of this thesis read and analyzed in detail the content of each one, seeking to find sentences that could characterize code smells in Elixir, whether traditional or specific.

All sentences were later validated by the advisor of this thesis, seeking to reach an agreement regarding the classification of a discussion as a code smell. Only eight artifacts out of 46 analyzed—M1, M2, M6, M7, M8, M22, M32, and M35—resulted in a lack of agreement between the thesis' author and his advisor.

An example of disagreement can be seen in M6. In this artifact, the developers started a discussion that could indicate a novel Elixir-specific code smell related to the update of values in a `Map`:

> *"Finding this typo was very hard because the* `|> Map.put` *syntax doesn't raise any compile errors. [It can be] a potential code smell. [...] Another way of updating [value] and throw an error if the key doesn't exist is to use the built-in* `update` *syntax [Instead of] using the* `Map.put` *function"*

In Elixir, the `Map.put(key, value)` function can be used to update a value associated with a key. However, if due to a typo, the given key does not exist, Elixir adds a new key to the `Map`, not accusing any error related to the non-existence of the key. On

[4] `https://github.com/Finbits`

[5] `https://github.com/elixir-lang`

[6] `https://github.com/phoenixframework`

GitHub, there was no consensus on whether this is a code smell. Also, among us, one reviewer understands that this problem is a Low-Level Concern smell, as it can confuse developers, causing the false sensation that the `Map` has been updated correctly, while the other reviewer thinks that this does not characterize a quality problem. Due to this disagreement, M6 was discarded.

In total, 39 code smells discussions were found in the 38 selected artifacts. As shown in Table 3.5, most discussions were found in Pull Requests and they refer more to Elixir-specific smells than to traditional smells.

Table 3.5: Code smell discussions by artifact type (MSR)

Artifact	**Traditional** *Discussions (#Novel)*	**Elixir-specific** *Discussions (#Novel)*
Source Code Files	0(0)	3(0)
Commits	2(0)	0(0)
Issues	1(0)	14(2)
Pull Requests	3(1)	16(2)
Total	6(1)	33(4)

Thus, only one novel Elixir-specific smell emerged from the MSR steps. This novel smell was found four times and expanded the catalog to a total of 23 specific code smells. Furthermore, a traditional code smell—Switch Statements—was also cataloged in this third prospecting approach.

3.1.2 Do Elixir developers discuss traditional code smells? (RQ1)

Using our mixed methodological approach, we found discussions about 12 traditional code smells, as shown in Table 3.6. Discussions found in our GL review are identified starting with the letter "G", while those from the interaction with the Elixir community are identified with the letter "E" and those found in our MSR approach are started with the letter "M".

One of the most discussed traditional smells is Comments. This smell was found in four sources, all from the grey literature. In G12, for example, the author argues that using comments to document code in languages that have a specific construct for documentation is a code smell:

"[...] for me documentation isn't a comment, in most languages [Unfortunately] documentation happens to be represented as a comment. [...] some languages, such as Elixir, Clojure and Rust, have a separate construct for documentation to make this obvious and facilitate working with documentation. [...]"

Table 3.6: Traditional code smells discussed by Elixir developers (RQ1)

Traditional smell	**Sources**	**#Sources**
Comments	G1, G10, G12, G14	4
Long Parameter List	G1, G16, E2, M3	4
Long Function	G1, E2, E7	3
Primitive Obsession	G3, M4, M13	3
Shotgun Surgery	G1, G17, M5	3
Duplicated Code	G1, M26	2
Feature Envy	G1, G6	2
Divergent Change	G1	1
Inappropriate Intimacy	G1	1
Large Class	G1	1
Speculative Generality	G1	1
Switch Statements	M10	1

Another code smell—Long Parameter List—was found in four discussions coming from the grey literature, GitHub artifacts, and interaction with the Elixir community. In G16, the author compares a way to remove Long Parameter List in Elixir with strategies to remove this smell in object-oriented languages:

"A long parameters list is one of many potential bad smells [...]. In object-oriented languages like Ruby or Java, we could easily define classes that help us solve this problem. Elixir does not have classes but because it is easy to extend, we can define our own types."

Primitive Obsession emerged from three different sources during our exploration. It was found mainly in our MSR approach, being the traditional smell most found by this strategy. In commit M4, although the author did not explicitly name this smell, he describes the replacement of variables of type `float` (primitive) by ones of a composite type:

"[...] Cleaned up some code smell [...] Deprecates use of 'floats' for money amounts... [Instead] Introduces the 'Gringotts.Money' protocol."

In Elixir, `protocols`[7] are polymorphic mechanisms similar to `interfaces` in Java. In this way, they can be used to extend primitive types. In commit M4, variables of the

[7] `https://elixir-lang.org/getting-started/protocols.html`

primitive type `float` were replaced by implementations of a protocol—`Gringotts.Money`—that more accurately represent the characteristics of money values.

Shotgun Surgery also emerged from three different sources in our research. In G17, the author refers to a problem where particular code modifications require many small changes in different files, which is the main characteristic of Shotgun Surgery:

> *"[When using microservices we] need to be able to deploy independently. [Despite that] tight coupling could be found through shared libraries forcing an upgrade throughout the system. Or [microservices] could be coupled through a database schema where many services need to upgrade after a schema change."*

Switch Statements is a traditional smell found only in our MSR approach. GitHub users who participated in M10's discussions clearly refer to a situation related to it:

> *"[...] When I see a type field, it's a little bit of a code smell: it usually ends with a "replace conditional with polymorphism" refactor. Different types of players will behave differently but conform to the same interface. That's a great case for polymorphism [...]"*

Using a parameter to inform a type for a function can decrease software quality. Considering these functions can be scattered throughout different modules, if in the future new types need to be handled, many conditionals will need to be updated in different parts of the code. These Elixir's control-flow structures are analogous to `switch` statements in other languages. According to Fowler and Beck [74], this smell can be removed through the use of polymorphism, as also suggested in M10.

By analyzing Table 3.6, we can conclude that 16 (out of 88) sources analyzed in our studies have discussions on traditional code smells. However, a single document (G1) concentrates most discussions. This document discusses all, except two, traditional smells that emerged from our methods. When comparing the results obtained by each method, 11 of the 22 traditional code smells (50%) were found in the grey literature review, two smells were discussed in the interaction with the Elixir community, and five were in our MSR approach.

> **RQ1 answer:** Traditional code smells are also important in modern functional languages like Elixir, as discussions about more than half of them (12 out of 22) were found in our study.

3.1.3 Do Elixir developers discuss other smells? (RQ2)

We found discussions about 23 Elixir-specific smells, and we classified those into two different groups, as described previously in our study design (Section 3.1.1). Table 3.7 summarizes the 14 smells classified as Design-Related, and Table 3.8 does the same for nine smells classified as Low-Level Concerns. We selected a subset of the code smells to present in more detail in this section. They were chosen because they jointly present an overview of the main features of their respective categories, thus enabling a good understanding of the catalog.

More details and examples on all 23 Elixir-specific smells can be found in our catalog on GitHub [195].

3.1.3.1 Design-related smells

In Elixir it is possible to overload a function by specifying different clauses composed of patterns and guard checks. These clauses are matched against the values of the arguments when the function is called to define which clause will be executed. These overloaded functions are known as multi-clause functions. Thus, Unrelated Multi-Clause Function is a Design-Related smell that poses a problem peculiar to Elixir. For example, according to G10's author, the abuse of this resource makes the code difficult to understand, as follows:

> *"In Elixir, we can use multi-clause functions to group functions together using the same name. [However] when we start adding and mixing more pattern matchings and guard clauses [...] trying to squeeze too many business logics into the function definitions, the code will quickly become unreadable and even harder to reason with. [...]"*

Complex Extractions in Clauses represent another significant smell of the Design-Related group. Usually, when we use multi-clause functions, it is possible to extract values from `structs` for further usage and for pattern matching or guard checking. These values can be used both in the function body and in its interface, therefore impairing code readability, making it difficult to trace the origin of the values, especially when we have many clauses or many arguments, as reported in E6:

> *"[...] Pattern matching [can be used] to extract fields [...]. Once you have too many clauses or too many arguments, it becomes hard to know which parts are used for pattern/guards and what is used only inside the body [...]."*

Table 3.7: Design-related Elixir-specific code smells (RQ2)

Elixir-specific smell	**Description**	**Sources**
Using App Configuration for Libraries	A library function that is configured using parameterization mechanisms instead of arguments, thus limiting its reuse by clients	G5, M18, M31, M36, M37, M38, M41, M42, M45
Using Exceptions for Control-Flow	A library that forces clients to handle control-flow exceptions	G5, G11, M31, M33, M40, M44
Code Organization by Process	Library unnecessarily organized as a process, instead of modules and functions	G5, M20, M29, M30
Unsupervised Process	When a library creates processes outside a supervision tree, therefore not allowing users to control their apps fully	G5, M14, M34, M43
"Use" Instead of "Import"	Module that relies on `"use"` to declare dependencies when an `"import"` is enough. Typically, `"use"` implies a tight coupling with the target module	G5, M19, M27, M46
Compile-Time Global Configuration	Function that uses module attributes for configuration purposes, therefore preventing run-time configurations	G5, M23, M39
Complex Extractions in Clauses	Function that uses pattern matching in its signature to extract values used both in guard checks and in its body	E6, M12
Large Code Generation by Macros	`Macros` that generate a lot of code, affecting compilation or execution performance	E7, M28
Unrelated Multi-Clause Function	Function with many guard clauses and pattern matchings	G10, M9
Agent Obsession	When the responsibility for interacting with an `Agent` process is spread across the system	G8
Data Manipulation by Migration	Module that performs both data and structural changes in a DB schema via `Ecto.Migration`	G9
GenServer Envy	Using a `Task` or `Agent` but handling them like `GenServers`	G8
Large Messages	Processes that exchange long messages frequently	G13
Untested Polymorphic Behaviors	Function with a generic `Protocol` type parameter, but that does not have guards verifying its behavior	G7

Listing 3.3 illustrates the occurrence of Complex Extractions in Clauses. The `drive/2` multi-clause function extracts from a `%User{}` struct the value of the field `name` for further usage (lines 2 and 5), and the value of the field `age` for pattern/guard checking (lines 1 and 4). In addition to the `%User{}` struct, the two clauses of this function also receive the boolean argument `d_lic` to define if the user has a driver's license. This value is then used in the function guard (`when` clauses), as follows.

Listing 3.3: Example of Complex Extractions in Clauses

```
1 def drive(%User{name: n, age: a}, d_lic) when a >= 18 and d_lic == true do
2   "#{n} can drive"
3 end
4 def drive(%User{name: n, age: a}, d_lic) when a < 18 or d_lic == false do
5   "#{n} cannot drive"
6 end
```

A solution to remove this smell is to extract in the function signature only values that are used in the function clauses, as in the Listing 3.4.

Listing 3.4: Refactoring of Complex Extractions in Clauses

```
1 def drive(%User{age: a} = user, d_lic) when a >= 18 and d_lic == true do
2   %User{name: n} = user
3   "#{n} can drive"
4 end
5 def drive(%User{age: a} = user, d_lic) when a < 18 or d_lic == false do
6   %User{name: n} = user
7   "#{n} cannot drive"
8 end
```

Our next discussed smell happens in parallel and distributed environments. In such environments, the unnecessary use of parallelization to organize code poses a problem in some contexts. Particularly, Code Organization by Process was the second most discussed smell in the Design-Related category. This smell occurs when a library uses processes unnecessarily, imposing a specific parallelization behavior on its clients.

Listing 3.5 illustrates an instance of this smell. In this example, the `Calculator` library implements arithmetic operations—`add/3` (line 4) and `subtract/3` (line 8)—through a `GenServer`, which is one of the process abstractions provided by Elixir. In lines 25-30, examples of using this code are presented. In line 26, the process responsible for running the code is started with the initial state zero. The `init/1` function (line 12), which is a `GenServer` callback, is automatically called when the `Calculator` process is started to set its initial state. However, this value is not used for any purpose in the code. Instead, the process identifier is used in the `add/3` and `subtract/3` to make calls to the `Calculator` process (lines 5 and 9) and then to wait for replies, which are provided respectively by the `handle_call/3` functions, implemented in lines 16 and 20. These functions always return a tuple composed of three values, the second one containing the operation result.

Listing 3.5: Example of Code Organization by Process

```
1 defmodule Calculator do
2   use GenServer
3
4   def add(a, b, pid) do
5     GenServer.call(pid, {:add, a, b})
6   end
7
8   def subtract(a, b, pid) do
9     GenServer.call(pid, {:subtract, a, b})
```

```
10    end
11
12    def init(initial_state) do
13      {:ok, initial_state}
14    end
15
16    def handle_call({:add, a, b}, _from, state) do
17      {:reply, a + b, state}
18    end
19
20    def handle_call({:subtract, a, b}, _from, state) do
21      {:reply, a - b, state}
22    end
23  end
24
25  # Starting the process that organizes the code
26  iex(1)> {:ok, pid} = GenServer.start_link(Calculator, 0)
27  {:ok, #PID<0.132.0>}
28  ...
29  iex(2)> Calculator.add(1, 5, pid)       # 6
30  iex(3)> Calculator.subtract(2, 3, pid)  # -1
```

Although the code works correctly, when a library uses a process to organize its code, its readability is reduced due to complex logic. In addition, this organization forces the clients to work with processes even when that would not be their design choice.

The most discussed Elixir-specific smell was Using App Configuration for Libraries (nine documents), which was classified as a Design-Related smell. In M45, for example, a GitHub user reports that using the *config.exs* file to parameterize values used inside a library is a code smell that makes a library less flexible, limiting its reuse. This quote illustrates some sentences taken from M45 that characterize this smell:

> *"[...] using config[.exs] for libraries is considered an anti-pattern [...] For instance, see this discussion where [...] Jose Valim and others discuss it and recommend, if possible, configuring your library through function arguments. [Using parameterized values to configure libraries] gets tricky setting the configuration with regard to compilation and releases."*

As illustrated in Listing 3.6, `DashSplitter` module is a library that configures the behavior of its functions through the Application Environment parameterization mechanism. `DashSplitter` implements `split/1`, a function to separate a string into a certain number of parts. The character used as a separator in `split/1` is always `"-"` and the number of parts the string is split into is parameterized. This parameterized value is retrieved by `split/1` in line 3. Therefore, all clients are forced to use `split/1` with the same number of parts. In our example, this value is three, as defined in the *config.exs* file (line 3).

Listing 3.6: Example of Using App Configuration for Libraries

```
1  import Config
2  config :g_config,
```

```
3 parts: 3
4 import_config "#{config_env()}.exs"
```

```
1 defmodule DashSplitter do
2   def split(string) when is_binary(string) do
3     parts = Application.fetch_env!(:g_config, :parts) # <= get config
4     String.split(string, "-", parts: parts)           # <= parts: 3
5   end
6 end
```

3.1.3.2 Low-level concerns smells

Dynamic Atom Creation is our first example of smell in the Low-Level Concerns category. An atom is a basic Elixir data type that is not collected by the language's garbage collector, so atoms live in memory throughout an application's execution cycle. However, BEAM (the virtual machine used by Elixir) has a limit on the number of atoms that can exist in an application. For this reason, the dynamic creation of atoms is considered a code smell. M16's author describes Dynamic Atom Creation as follows:

> *"[...] creating atoms from untrusted input is bad practice since atoms are not garbage collected. In addition, the number of atoms is limited to 1,048,576 and the size of each atom can be at most 255 [...]"*

As a second example, Complex else Clauses in with is a Low-Level Concerns smell that refers to `with` statements that flatten all their error clauses into a single complex `else` block. Listing 3.7 illustrates an instance of this smell, where the function `open_decoded_file/1` reads a base 64 encoded string content from a file (lines 2-3) and returns a decoded binary string (line 4). This function has an `with` statement that handles two possible errors in a single `else` block (lines 5-8). Although this example has an `else` block that handles only two errors, as this function evolves it can gain new types of errors, which will harm readability and maintainability.

Listing 3.7: Example of Complex else Clauses in with

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

Table 3.8: Low-level concerns Elixir-specific code smells (RQ2)

Elixir-specific smell	**Description**	**Sources**
Dynamic Atom Creation	Function that creates `atoms` in an uncontrolled and dynamic way, affecting memory management.	M16, M17, M23, M25
Working with Invalid Data	A function that does not validate its parameters, potentially inducing a caller to introduce a hard-to-understand bug	G5, M11, M21
Unnecessary Macros	Using `macros` instead of functions and `structs`	G5, M15
Accessing Non-Existent Map/Struct Fields	Accessing `struct` or `map` fields dynamically may result in null values that can be ambiguous and lead a programmer to introduce bugs	G7
Alternative Return Types	Functions with parameters that significantly change their return type	E3
Complex Branching	Function that handles multiple errors, making it complex	G2
Complex else Clauses in with	`with` statements that handle all their error clauses in a single `else` block	E4
Modules with Identical Names	Modules with identical names, preventing their simultaneous load	G5
Speculative Assumptions	Code that makes assumptions that have not been planned for, thus returning incorrect values when a crash is desired	G7

Another Low-Level Concerns smell, Working with Invalid Data, has been discussed both in the grey literature review and in the MSR approach. G5's author describes this smell as follows:

> *"Elixir programs should prefer to validate data as close to the end-user [...] so the errors are easy to locate and fix. [When this is not done] if the user supplies an invalid input, the error will be raised deep inside [the function], which makes it confusing for users. [...] when you don't validate the values at the boundary, the internals [of the function] are never quite sure which kind of values they are working with."*

RQ2 answer: Using three different research methods, we were able to find 23 Elixir-specific code smells. Among them, nine are from the Low-Level Concerns category and 14 are Design-Related smells.

3.1.4 Threats to Validity

Construct Validity: The main threat to construct validity is related to the format of the search queries used in our grey literature review (GLR) and in our mining software repository (MSR) methodological approach. If some keyword combinations are missing in queries, the search results can include many artifacts out of context or even important documents cannot be retrieved. To mitigate this threat, as proposed by Garousi *et al.* [76], we performed preliminary searches to calibrate the queries, adding code smells synonyms, both in singular and in the plural, and deleting some keywords that were polluting the results.

Conclusion Validity: Since our code smell prospection is based on a grey literature review, interaction with the Elixir community, and MSR approach, our sources are documents that were not peer-reviewed. To reinforce the validity of our results, we carefully inspected the documents and artifacts returned by executing the queries or created by the Elixir community. Both the author of this thesis and his advisor did a preliminary reading of all sources, selecting for further analysis only those that are relevant and are written by professionals who work or have experience with Elixir. However, as our opinion was a fundamental factor in defining which documents would be analyzed and which problems extracted from them should be cataloged as a code smell, there was a risk of bias in these qualitative analyses. To prevent the results from representing only an individual view and being biased by personal experiences, the author of this thesis and his advisor participated in these methodological steps. Thus, the artifacts selected for in-depth analysis represented the agreement of both regarding the relevance and quality of the results.

Internal and External Validity: The threats here concern the degree to which we can generalize the relevance of the catalog of code smells proposed by this work. This risk was partially mitigated, since we used a mixed methodology of code smells prospection, having as sources documents from the grey literature with heterogeneous representativeness (individual or companies views), spontaneous contributions from the Elixir community coming from developers with distinct backgrounds, and GitHub public repositories with different sizes, purposes, domains, and popularity. Ultimately, we mitigated this risk more broadly by conducting a survey with Elixir developers. This survey, which will be presented in the next section (3.2), validates these smells from the perspective of developers.

3.2 Catalog Validation

To validate the catalog of code smells for Elixir presented in Section 3.1, we conducted a survey with developers who use this language. Particularly, in this current section we present a study that collected quantitative data with this instrument to answer the following research question:

RQ3. What are the developers' perceptions of code smells in Elixir? In this RQ, we seek to understand, from the developers' perspective, the prevalence and relevance levels of each of the traditional and Elixir-specific smells in real projects. Our definition of relevance, as communicated to the participants in the survey form, relates to the potential that a smell has to produce negative impacts on the maintainability, comprehensibility, and evolution of the code.

As this validation involves the direct participation of human subjects, before conducting this survey, it was evaluated and approved by the Research Ethics Committee of the Federal University of Minas Gerais, Brazil (CAAE: 58679222.8.0000.5149).

We dedicate Section 3.2.1 to present the methodology applied to design, conduct, and analyze the results of our survey. In Section 3.2.2, we present and discuss the results on the prevalence and relevance of code smells in Elixir. Finally, Section 3.2.3 discusses threats to validity.

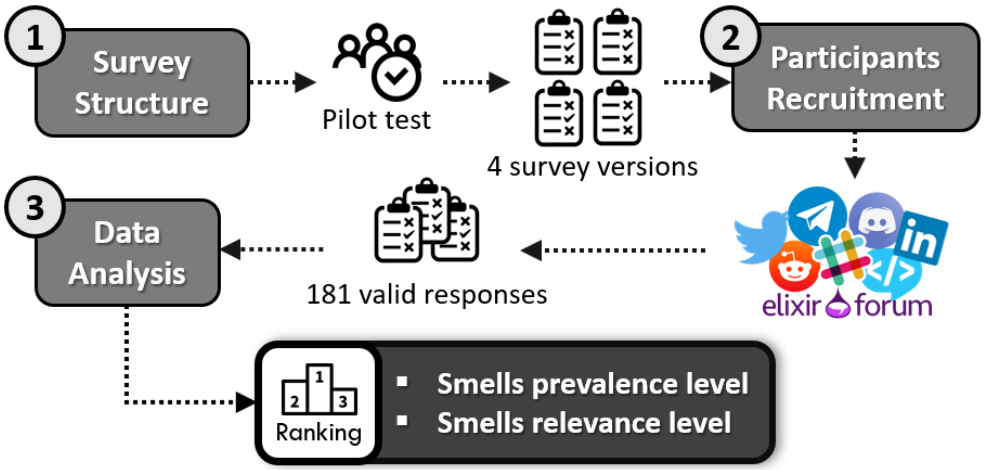
3.2.1 Survey Design

Although the code smells for Elixir have been prospected from documents and code artifacts created by professionals who work with this language, we decide to conduct a survey to reveal the extension of the impacts caused by each of these sub-optimal code structures in the daily lives of developers. Figure 3.2 summarizes the steps we followed to conduct this survey. We also detail these steps in the following paragraphs.

1) Survey Structure: We began our survey questionnaire by explaining the goals of the study, introducing the research team, describing the voluntary nature of participation, and getting consent from the participants (Appendix A.1). All the questions[8] that were asked to the participants during the study are available online in the replication package of this chapter [198] and in Appendix A.2. Particularly, the questions were organized

[8]Readers can also quickly check all these questions at: https://doi.org/10.5281/zenodo.7430258

Figure 3.2: Overview of survey on code smells in Elixir



according to the following topics:

– *Demographics*: In this topic, we asked about the participant's geographic location, their number of years of experience with Elixir, and the number of Elixir projects they have worked on.

– *Perceptions on code smells in Elixir*: After presenting a list of traditional and Elixir-specific smells, accompanied by their respective descriptions and code examples, we asked the participants how often they encounter each smell in the Elixir projects they work on and what is the level of the negative impact of each smell on the maintenance and evolution of these projects. The responses were given on scales of one (*very low*) to five (*very high*). To prevent the order in which the smell is presented influence the quality of the responses, they were presented in random order. Finally, to ensure the quality of our results, all questions that involved developers' perception of code smells were optional, thus avoiding forcing participants to answer about smells for which they did not have sufficient background knowledge.

During this elaboration step, we invited seven developers from our network who work with Elixir to participate in a pilot test. Five of these developers answered the pilot test and gave us feedback that helped us to clarify some questions and mainly to reduce the size of the questionnaire, making it faster to be answered.

Initially, we planned to ask each participant their perception of each of the 35 code smells in our catalog. During the pilot test, we realized that the participants were taking a long time to complete the questionnaire and therefore we decided to create four different versions of the survey, each one asking the participant's perception of a maximum of nine code smells. In this way, each smell was included in only one version of the survey, and the selection of the smells in each of the versions was made randomly. Table 3.9 presents the distribution of the smells in the survey versions.

Table 3.9: Survey versions template

Version	# Smells by type
A	3 traditional smells + 6 Elixir-specific smells (2 *LL* + 4 *DR*)
B	3 traditional smells + 6 Elixir-specific smells (2 *LL* + 4 *DR*)
C	3 traditional smells + 6 Elixir-specific smells (3 *LL* + 3 *DR*)
D	3 traditional smells + 5 Elixir-specific smells (2 *LL* + 3 *DR*)

LL: Low-level concerns. / *DR*: Design-related.

Each of the survey versions was constructed using a different Google form. To define which version each participant should answer, we implemented a small algorithm that generates a random number in the range from zero to three. According to the number generated, this script automatically forwards the participant to one of the survey versions.

The pilot test answers were solely used to validate and receive feedback on our survey design procedures. Thus, we do not consider these answers in our data analysis.

2) Participants Recruitment: In this step, aiming to increase the representativeness of our results, we focused to recruit a sufficient number of Elixir developers with multiple levels of experience, coming from different cultures and working on various projects.

Instead of sending messages to developers using emails collected at GitHub [17], we followed a recruitment approach based on public social media. First, we created a post on Twitter inviting Elixir developers to respond to our survey and share it with their private networks. In this tweet we use the most common hashtags from the Elixir developer community,[9] thus increasing the reach potential of our publication. At the same time, we also promoted the survey on the Elixir Forum, the official discussion platform for the developers of this language. Second, throughout the two weeks of the survey, every two days we made publications inviting participants on the language's official channels at Discord, Slack, Telegram, DevTalk, Reddit, and LinkedIn (in a group with 7K members).

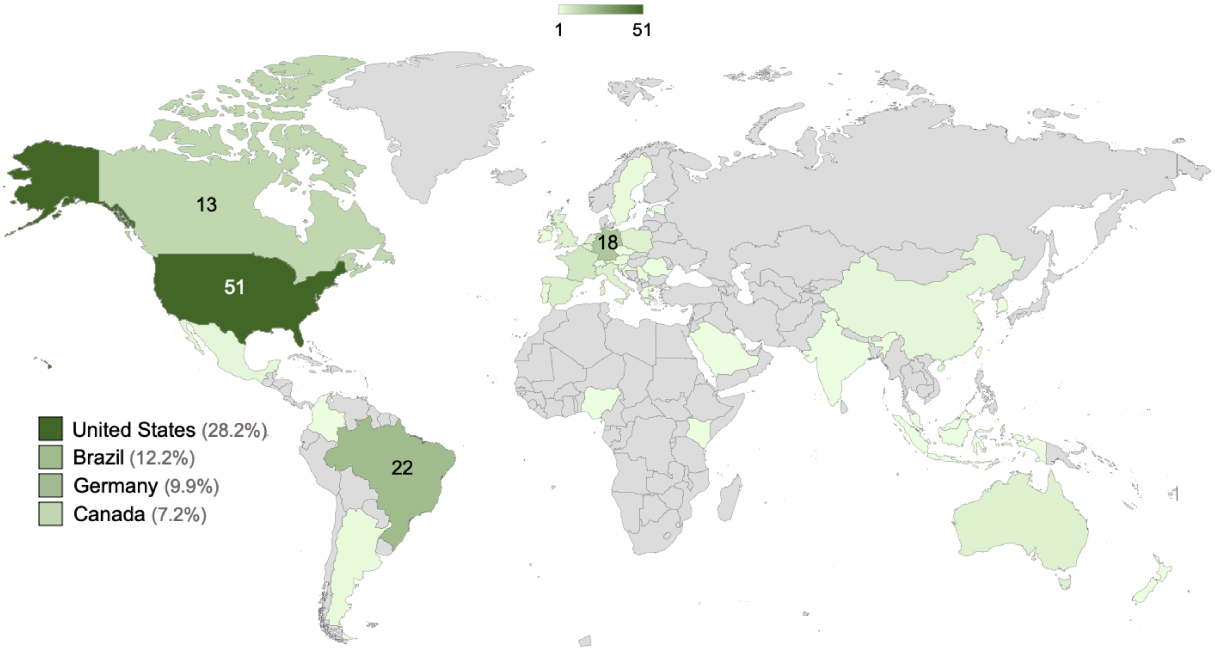
After two weeks, we closed the survey with 182 responses. We analyzed all these responses to filter out those provided by participants who are also authors of documents used in the previous steps of this work to prospect code smells (Section 3.1). After this analysis, we found that only one survey respondent is also an author of a document used in the grey literature review. Therefore, we removed this response to completely avoid possible biases, resulting in 181 valid responses.

The survey participants are distributed across 37 countries and all continents. As shown in Figure 3.3, the majority of our participants currently work in America (51%) and Europe (40%), with the United States (28%) and Brazil (12%) being the top two countries. In terms of experience with Elixir, nearly 69% of our participants have more than three years of experience, and 63% have worked on more than four different projects

[9] #MyElixirStatus and #ElixirLang

using Elixir.

Figure 3.3: Countries where the survey participants reside



Note: The legend presents the top 4 countries with most participants.

3) Data Analysis: To analyze our responses, we used descriptive statistics. As the number of responses for each of the survey versions varied (as can be seen in Table 3.10), we normalized the prevalence and relevance of each code smell by calculating the arithmetic mean of all responses for these questions. Thus, the closer to five the prevalence or relevance level of a code smell is, we can consider it more common or harmful for Elixir systems respectively.

Table 3.10: Responses for each survey version

Version A	Version B	Version C	Version D
27	38	58	58

Although questions regarding the relevance and prevalence of each code smell were optional, when we analyzed all smells evaluated in our survey, the average percentage of valid (*i.e.,* non-empty) answers was 99.1%, with a standard deviation of 1.5%. The question with the lowest percentage of responses (94.8%) was about the relevance of the Unsupervised Process smell, where only three of the 58 respondents chose not to answer.

To assess whether the respondents' level of experience influenced their perception of the prevalence and relevance of smells, we used the Mann-Whitney test [167]. This non-parametric test was chosen because our data did not show a normal distribution. Two distinct grouping variables were used in this test, namely the number of years a

participant has worked with Elixir (*at most three* or *more than three*) and the number of different projects they have participated in using this language (*at most four* or *more than four*). For all analyses, we used the SPSS statistical analysis tool,[10] and considered a significance level of 0.05.

3.2.2 What are the developers' perceptions of code smells in Elixir? (RQ3)

As can be seen in Figure 3.4, we chose to present the results using a scatter plot to facilitate the comparison between smells. In this plot, we also decided to divide the plane into nine zones to show the perceived relevance and prevalence levels for each smell. Particularly, in each axis, there are three quadrants: LOW (average scores between 1.0 and 2.3), MID (average scores between 2.3 and 3.7), and HIGH (average scores between 3.7 and 5.0).

The scatter plot reveals that the three code smells with the highest levels of relevance are respectively UNNECESSARY MACROS (UNM, 4.16),[11] DYNAMIC ATOM CREATION (DAC, 3.82), and WORKING WITH INVALID DATA (WID, 3.81). In contrast, the three least relevant smells are COMMENTS (COM, 2.29), DATA MANIPULATION BY MIGRATION (DMM, 2.46), and "USE" INSTEAD OF "IMPORT" (UII, 2.55). As can also be seen in the figure, the only smell with a low-relevance level is COMMENTS, all the others have relevance levels at least in the MID zone and 17.1% (6 out of 35) have HIGH relevance. In this way, most smells have the potential to cause non-negligible impacts on the maintainability, comprehensibility, or evolution of Elixir systems.

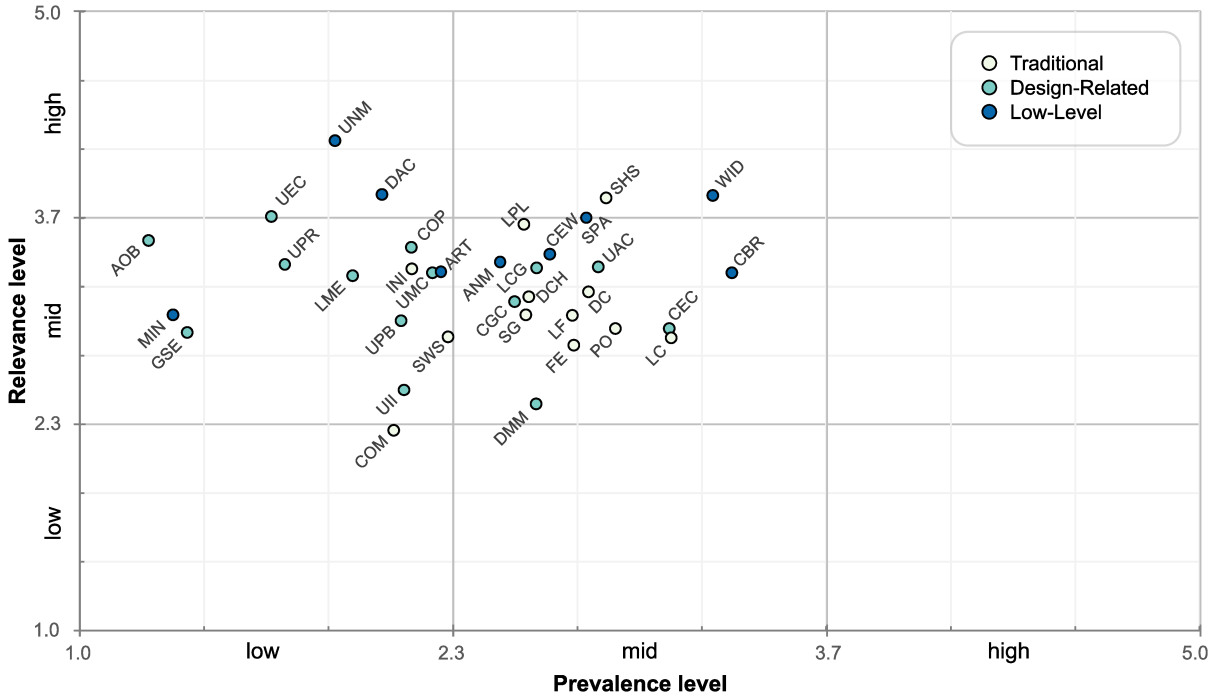
Comparatively, our data suggest that the negative impacts caused by Elixir-specific smells are greater than those caused by traditional smells, as the average relevance level of Elixir-specific smells was 3.32 compared to 3.09 for traditional smells. This is also evidenced when we specifically observe the smells evaluated with HIGH relevance by developers, as five out of the six smells that received this evaluation are Elixir-specific smells. Additionally, 11 out of the 13 code smells evaluated with MID relevance but in the range closest to the HIGH relevance zone are also Elixir-specific smells.

Regarding their prevalence, most smells (54.3%) have a mid-prevalence level. This shows that code smells are not uncommon in Elixir systems and therefore require attention. According to the survey participants, COMPLEX BRANCHING is the most prevalent

[10] `https://www.ibm.com/spss`

[11] When discussing each smell, we are adding between parentheses the acronym used in Figure 3.4 (*e.g.,* UNM) and the respective average score of the survey answers (*e.g.,* 4.16).

Figure 3.4: Developers' perception of code smells in Elixir (RQ3)



ANM : Accessing Non-existent Map/Struct Fields
ART : Alternative Return Types
CEC : Complex Extractions in Clauses
CGC : Compile-time Global Configuration
COP : Code Organization by Process
DC : Duplicated Code
DMM: Data Manipulation by Migration
GSE : GenServer Envy
LC : Large Class
LF : Long Function
LPL : Long Parameter List
PO : Primitive Obsession
SHS : Shotgun Surgery
SWS : Switch Statements
UAC : Using App Configuration for Libraries
UMC : Unrelated Multi-clause Function
UPB : Untested Polymorphic Behaviors
WID : Working with Invalid Data

AOB : Agent Obsession
CBR : Complex Branching
CEW: Complex Else Clauses in With
COM: Comments
DAC : Dynamic Atom Creation
DCH : Divergent Change
FE : Feature Envy
INI : Inappropriate Intimacy
LCG : Large Code Generation by Macros
LME : Large Messages
MIN : Modules with Identical Names
SG : Speculative Generality
SPA : Speculative Assumptions
UEC : Using Exceptions for Control-flow
UII : "Use" Instead of "Import"
UNM: Unnecessary Macros
UPR : Unsupervised Process

smell (CBR, 3.33). Its prevalence level is MID, although it is close to the lower limit of the HIGH prevalence zone (see Figure 3.4). The other two most prevalent smells are respectively WORKING WITH INVALID DATA (WID, 3.26) and LARGE CLASS (LC, 3.11). In contrast, the three least prevalent smells are AGENT OBSESSION (AOB, 1.25), MODULES WITH IDENTICAL NAMES (MIN, 1.33), and GENSERVER ENVY (GSE, 1.38). Differently from what we observed regarding the relevance level, our data suggest that traditional smells occur more frequently in Elixir systems, since the average prevalence level of traditional smells was 2.64, compared to 2.29 for Elixir-specific ones.

Most of the smells are concentrated in the mid-relevance level zones (28 out of 35). Among the nine zones of the plan, the one with the highest concentration is the central one (mid-prevalence and mid-relevance), resulting in a cluster composed of approximately 46% of the smells in the catalog.

Finally, to better understand which smells stood out the most in our catalog, we calculated the arithmetic mean of each smell between its relevance and prevalence levels. The three smells with the highest averages are respectively Working with Invalid Data (WID, 3.53), Shotgun Surgery (SHS, 3.34), and Complex Branching (CBR, 3.32), being, therefore, those that deserve special attention from Elixir developers.

Since the prevalence and relevance levels of each code smell, shown in Figure 3.4, were calculated using the arithmetic mean of their respective responses, we also decided to compute the standard deviation (SD) for both prevalence and relevance. This helps us better understand the dispersion of responses in relation to the averages. The SDs for prevalence range from 0.54 to 1.43, with Agent Obsession (AOB) having the lowest variation and Compile-time Global Configuration (CGC) showing the most heterogeneous responses. For relevance, the SDs fall between 0.97 and 1.54, where Long Parameter List (LPL) stands out with the most homogeneous responses, while Modules with Identical Names (MIN) exhibits the highest variation. Detailed values for standard deviation for each code smell are available in the replication package of this chapter [198].

The level of experience of the developers influenced their perception of just five code smells from our catalog. Specifically, the number of Elixir projects the developers worked on affected their perception of four out of the 35 smells, as shown in Table 3.11. Out of these four smells, two had a higher perceived relevance (Dynamic Atom Creation and Feature Envy) among developers who worked on a maximum of four different projects. On the other hand, developers who worked on more than four projects perceived that the other two smells (Primitive Obsession and Unrelated Multi-clause function) are more prevalent.

Table 3.11: Influence of the number of Elixir projects in the developer's perception of a smell

Smell (Perception)	**Mean rank**		**Significance**
	$\leq$ 4 projects	**$>$ 4 projects**	
Dynamic Atom Creation (R)	25.73	16.26	0.011
Feature Envy (R)	25.04	16.62	0.025
Primitive Obsession (P)	21.64	32.40	0.020
Unrelated Multi-clause function (P)	24.24	32.96	0.044

R: Relevance. / P: Prevalence.

The number of years working with Elixir has less influence on developers' perception of code smells in our catalog. As shown in Table 3.12, the prevalence of only two smells was affected by this factor. In both cases, developers with more than three years of experience found these smells to be more prevalent than less experienced ones.

The complete results with the prevalence and relevance levels of each of the 35 code

Table 3.12: Influence of the Elixir experience time in the developer's perception of a smell

Smell (Perception)	**Mean rank**		**Significance**
	$\leq$ 3 years	**$>$ 3 years**	
Compile-time Global Configuration (P)	19.41	33.35	0.004
Unrelated Multi-clause function (P)	21.25	32.64	0.016

P: Prevalence.

smells that compose our catalog, as well as the complete results of the Mann-Whitney test (including those without statistical significance), are available in the replication package of this chapter [198]: `https://doi.org/10.5281/zenodo.7430258`.

RQ3 answer:

- Most of the smells in Elixir have relevances between MID and HIGH (97%), therefore having the potential to cause non-negligible impacts on maintenance.
- Most of the smells in Elixir have a mid-prevalence level (54%), so they require attention from developers because they are not rare in production code.
- The scatter plot quadrant with the largest number of smells is the central one, indicating that almost half (46%) of the smells require the attention of Elixir developers both to prevent them and to remove them.
- The developers' level of experience influenced their perception of only five code smells from our catalog.

3.2.3 Threats to Validity

Construct Validity: Threats here concern the mapping between theory and the real world. The main threat of this kind in our study concerns the recruitment of representative participants to answer the survey. By recruiting participants who are inexperienced in software development, or even with little knowledge of Elixir, we could obtain results that are not consistent with reality. To minimize this threat, we promoted our survey especially on the language's official communication channels, in order to reach participants with the profile desired by our research. In addition, we checked the experience level of these participants and the number of different projects they have worked on using the language. Most participants (69%) have more than three years of experience in Elixir, and 63% have worked on more than four different projects using this language.

Conclusion Validity: The main threat of this kind in our survey refers to the different number of responses received by each version of the questionnaire. As each participant had to answer only one of the four versions of the questionnaire and this version allocation was done randomly, some versions had more responses than others. However, the total number of valid responses obtained was high (181) and the version of the questionnaire that received fewer responses had 27 participants, which is still a significant number for this type of empirical research. To compensate for the unbalance between our four groups, we normalized the responses for each of them. Finally, another threat of this kind concerns our participants' recruitment strategy. As we used a public social media approach, it was possible to receive biased responses from participants who were also authors of documents we used to catalog smells. To mitigate this threat, we eliminated the only response received from an author of a previously used document. Furthermore, among all the documents used to catalog the code smells, we were unable to identify the author of a single one, as it is part of the official Elixir documentation.

Internal Validity: This threat concerns the internal factors that could influence the study results. Rating the prevalence and relevance of a code smell on a scale of one to five can be subjective. To mitigate this possible subjectiveness, we included a brief explanation of the scale in the questionnaire. We also attached to each smell a description of the problems caused, and code examples containing these sub-optimal structures. Additionally, we offered participants the choice of not responding to questions about smells that they did not have enough background knowledge about. As reported by Nardone *et al.* [133], participant fatigue when answering a long questionnaire is another factor that could influence our results. To mitigate this threat, we distributed the 35 code smells into four different versions of the questionnaire, thus reducing the number of responses requested from each participant. Finally, the order in which the smells were presented to the participants was random, thus avoiding the threat that the first smells were privileged with better answers than the latter.

External Validity: The threats here are related to the generalization of our results, as the perception collected by our questionnaire regarding smells could be affected by some kind of bias from our group of participants, thus not representing the general perception of Elixir developers. To mitigate this threat, we widely disseminated our survey on various official channels of the Elixir community, seeking to recruit a heterogeneous group of developers, from diverse cultures and professional environments. This strategy made it possible to recruit participants from 37 countries and from all continents.

3.3 Implications

Based on the results presented in this chapter, we shed light on the following practical implications:

1. Priority in preventing code smells. Our results can guide developers that work with Elixir in their decision-making about which code smells they should pay more attention to avoid inserting into their code while programming. The prevalence level of code smells is a good indicator to consider in this decision, because the higher the prevalence of a smell, the more common it is in Elixir systems. As these smells are inserted more frequently, developers can be more careful to avoid inserting them in their systems.

2. Priority in refactoring code smells. When developers detect instances of different types of code smells simultaneously in their code, they may wonder which smell to remove first. In line with the previous implication, we conjecture that the relevance level is a good indicator to define the priority of refactoring smells, because the more relevant a smell, the greater is its potential to have a negative impact on a system. Thus, it is recommended to remove it before the others.

3. Directing efforts to adapt tools for automatically detecting code smells. As shown in our preliminary study [196], only three code smells from our catalog are automatically detected by Credo,[12] which is currently the most popular static code analysis tool for Elixir. Our catalog of code smells not only represents a good opportunity to power the capacity of tools to detect code smells but can also direct the efforts of the developers of these tools, indicating which are the most prevalent and relevant smells, therefore being the ones that should have detection strategies implemented first.

3.4 Final Remarks

In this chapter, we proposed and validated the first catalog of code smells for Elixir. We used a mixed methodology, based on a grey literature review, interactions with the Elixir community, and mining GitHub code repositories to prospect and document smells. Specifically, a total of 60 grey literature documents, 25 documents generated through interactions with the developer community—13 Issues and 12 Pull Requests—and 301 artifacts mined from Elixir code repositories on GitHub were analyzed to catalog 35

[12] http://credo-ci.org/

code smells that occur in Elixir systems.

The proposed catalog of code smells was validated by conducting a survey with 181 experienced Elixir developers. These developers, who come from 37 countries and all continents, expressed their perceptions regarding the prevalence and relevance of the smells that make up our catalog.

It is important to acknowledge that we do not claim the proposed catalog to be exhaustive. While we have aimed to provide a comprehensive compilation, we recognize that the catalog could be expanded through alternative methodologies for identifying smells or even by revisiting the same methods employed in this study at a later time, when new discussions may have emerged within the developer community.

We summarize the contributions of this chapter as follows:

- We cataloged 23 novel Elixir-specific code smells and categorize them into two groups, Low-Level Concerns smells (9) and Design-Related smells (14).
- We find that at least 12 traditional code smells (as proposed by Fowler and Beck [74]) are also present in Elixir systems.
- We showed that the majority of cataloged smells have at least mid relevance, and therefore have the potential to impair the readability, maintenance, and evolution of Elixir systems.
- Furthermore, we have shown that the occurrence of code smells in Elixir systems are common.

These findings have practical implications for both developers and researchers. Given the variety of code smells in Elixir, their non-rarity in production code, and that they have significant potential to negatively impact systems, it is necessary attention to both prevent the insertion of these sub-optimal code structures and also to conduct their removal when identified.

Replication Package. We provide the complete dataset used in this chapter and a replication package at: `https://doi.org/10.5281/zenodo.7430258`.

Chapter 4

Refactorings in Elixir

In this chapter, **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 [74], which are also compatible with Elixir code. To accomplish this, similar to what we did in our investigation on code smells (Chapter 3), we conducted two studies, with the first one focused on prospecting and documenting refactorings for Elixir, and the second one aimed at validating them with developers.

Similar to Fowler [74], this thesis **defines behavior preservation in refactorings as the principle that the external behavior of code must remain unchanged during the transformation process**. In other words, to preserve behavior, a refactoring strategy should only improve the internal structure of the code without altering how it works from the end user's perspective. This implies that all observable features of a program—including inputs, outputs, interactions with the external environment, and even existing bugs—must remain consistent before and after refactoring. Consequently, if a test suite adequately covers the refactored code using one of the strategies in our catalog, it is possible to confirm that the changes do not introduce regressions.

This chapter is organized as follows. In Section 4.1, we outline our catalog of Elixir refactorings. Also, we detail our mixed methodology, based on a systematic literature review, a grey literature review, and on the mining of artifacts in GitHub repositories to prospect and document new refactorings for Elixir. Additionally, this section presents potential threats to validity concerning the identification and documentation of our refactorings. In Section 4.2, we delve into the details of the survey we conducted with Elixir developers to validate our catalog of refactorings for Elixir. This section presents the survey findings, discusses potential validity issues, and describes the questionnaire design and the methods used to recruit respondents. In Section 4.3, we discuss the practical implications of our findings. Finally, we conclude this chapter in Section 4.4.

4.1 Catalog of Refactorings for Elixir

In this section, we introduce the first study of this chapter, 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 [106, 113, 119, 156], Haskell [36, 105, 106, 111], OCaml [153, 154], among others, to the best of our knowledge, only one preliminary study conducted by us [200] 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 [200] 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 [74] are important in the Elixir context, but also to identify if other refactorings typical of functional languages or even specific refactorings occur in Elixir.

We dedicate Section 4.1.1 to present the mixed methodology applied to proposing a catalog of refactorings for Elixir. Next, in Section 4.1.2, 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 4.1.3 discusses threats to validity.

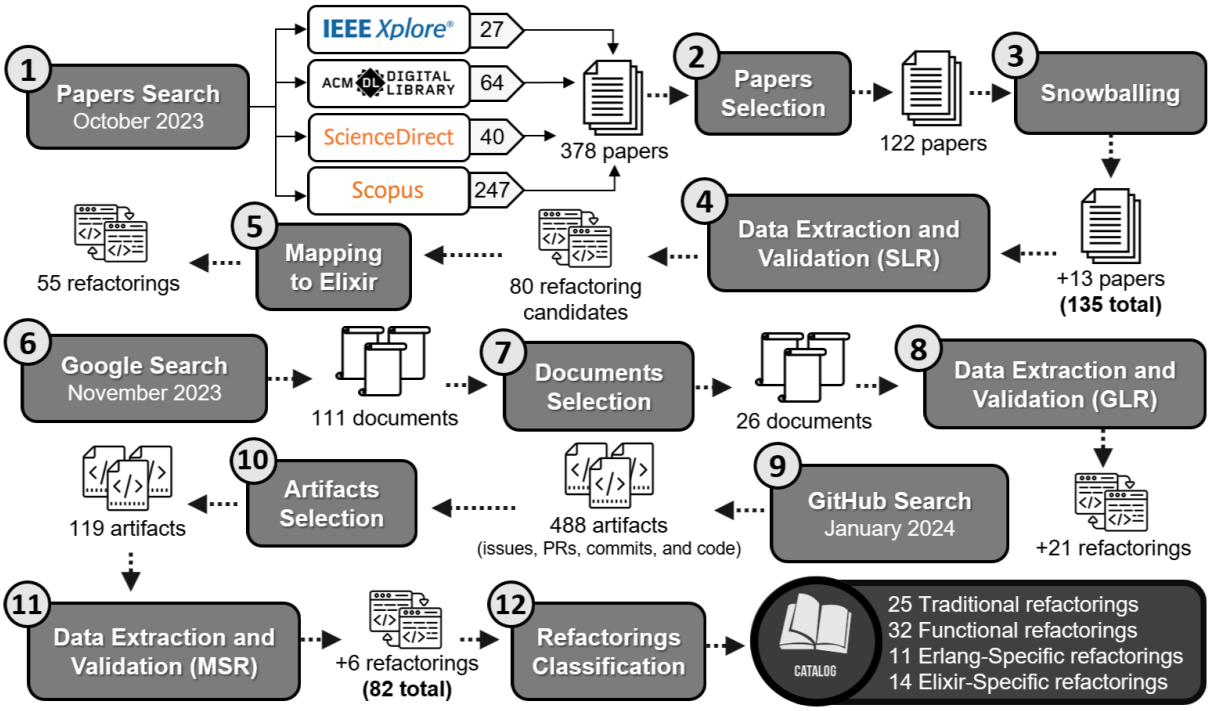
4.1.1 Study Design

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 4.1 summarizes our steps to propose the catalog of refactorings for Elixir. We also detail these steps in

the following paragraphs.

Figure 4.1: Overview of methods for cataloging refactorings in Elixir



1) Papers Search: In this step, we begin a systematic literature review to uncover refactorings compatible with code implemented in Elixir. To do this, we based on specific guidelines for conducting SLR studies in software engineering [18, 41, 99], 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 [99] 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 4.1.

Listing 4.1: 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 4.1 was carried out by analyzing the Top-100 most popular programming languages according to the TIOBE index.[1] The 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 was subsequently validated by the advisor of this thesis, 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,[2] ACM digital library,[3] ScienceDirect,[4] and Scopus.[5] We also adapted the query string to formats compatible with these digital libraries and the final search was conducted in October 2023.

2) Papers Selection: A total of 378 papers, labeled P1 to P378, were found after conducting searches across the four digital libraries chosen in the previous step. The abstracts of all papers were reviewed by the author of this work, 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 [207] for conducting snowballing in systematic reviews of software engineering studies. Therefore, the author of this work 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

[1] `https://www.tiobe.com/tiobe-index/`
[2] `http://ieeexplore.ieee.org`
[3] `https://dl.acm.org`
[4] `http://www.sciencedirect.com`
[5] `https://www.scopus.com`

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 author of this study 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 advisor of this thesis. However, some of these candidates represented the same transformation but with different names. For example, as shown in Table 4.1, nine variations of the refactoring candidate Rename an identifier were found.

Table 4.1: 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 author of this work used thematic analysis, a technique for identifying patterns (or "themes") within a collection of data [52, 53, 169] to group these candidates and then name the groups. At the end of this process, 80 different candidates were identified in this step by the author and validated by the advisor of this thesis. 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.

5) Mapping to Elixir: In this step, all 80 refactoring candidates extracted from our SLR were analyzed by the author of this work 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[6] and SD Erlang,[7] 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 author of this study produced code examples in Elixir to illustrate and document all these code transformation

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

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

strategies. Additionally, we provide tailored documentation of side conditions for each refactoring in Elixir, defining conditions that need to be respected before and after the transformations so that they do not alter the behavior of the refactored code.

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 [96, 196, 199, 214], 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.* [76]. 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 4.2 presents our query after this calibration.

Listing 4.2: GLR query string

```
1  ("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 4.1) 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 [142], 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 study on code smells for Elixir (Chapter 3), 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 [196, 199]. This requirement to limit the number of documents to be analyzed is

recurring in GLRs, as described by Garousi *et al.* [76].

We also replicated the validity criteria used in our previous study (Chapter 3), which is an adaptation of the Quality Assessment Checklist proposed by Garousi *et al.* [76]. Therefore, the documents were analyzed by the author and the advisor of this thesis in the order returned by Google, and to be considered valid, first, the documents should not be authored by them. 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?[8] (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 author of this work,[9] 31 were not related to our RQ, and two did not meet the authority criteria. Table 4.2 shows the distribution of the 26 valid documents among the three authority criteria.

Table 4.2: 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

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,[10] Veeps,[11] Trybe,[12] Plataformatec,[13] among others. Additionally, documents from the official Elixir documentation or authored by companies working with Elixir, such as DockYard,[14] 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 author of this study to find discussions about refactoring strategies performed in Elixir. Subsequently, the advisor of this thesis validated the decisions made regarding extracting or not refactoring strategies from these

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

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

[10]`https://remote.com/`

[11]`https://veeps.com/`

[12]`https://betrybe.com/`

[13]`https://plataformatec.com/`

[14]`https://dockyard.com/`

documents. 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:

> *"[Trying to refactoring] 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.

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 [52, 53, 169], 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.* [56] 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).

However, instead of conducting an open search in all existing repositories on GitHub, which could generate a large and polluted volume of artifacts, we selected the Top-10 Elixir repositories with the most stars on GitHub and then mine artifacts only from them. In this selection, repositories that contain only documentation (*e.g.,* Awesome Elixir[15]) were disregarded. Table 4.3 lists the repositories that were the subject of study in our MSR.

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 4.2), as shown in Listing 4.3.

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

Listing 4.3: MSR query string

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

We executed this query in January 2024 for each of the 10 selected repositories (Table 4.3), 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.

Table 4.3: 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.

10) Artifacts Selection: As shown in Table 4.3 and Table 4.4, 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.

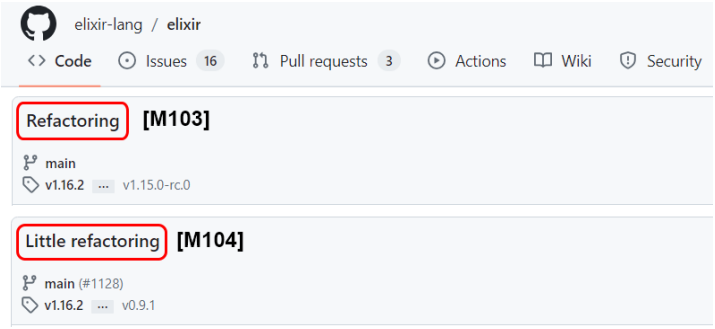
To select these 119 artifacts, all 488 retrieved artifacts were analyzed by the author and subsequently validated by the advisor of this thesis, aiming for agreement on the fulfillment of the following selection criteria. First, to be selected, an artifact should align with the context of our research question (RQ1). Second, the artifact cannot have been previously retrieved in our GLR or authored by the author or advisor 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 4.2 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 author of this work.[16] Table 4.4 provides an overview

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

Figure 4.2: Examples of unclear commits



of the retrieved and selected artifacts organized by types. The most retrieved and selected artifact type was the Pull Request, where 24.7% of retrievals were selected. In contrast, the selection rate of Files was only 12%.

Table 4.4: 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

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 author of this work read and analyzed these artifacts thoroughly. Unlike the process used to select artifacts, for extracting the refactorings, the 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 advisor of this thesis, seeking to reach an agreement on whether they represent a refactoring or not. The author and his advisor 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`,[17] code comments, and pipe indentation in Elixir:

> *"[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."*

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

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.

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 4.5.

Table 4.5: Categories used to organize the catalog of refactorings for Elixir

Category	**Description**	#	%
Traditional	Refactoring strategies described on Fowler's catalog [74]	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

4.1.2 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 and their corresponding code examples is available in our GitHub public repository [197].

4.1.2.1 Functional refactorings

This category had the highest number of Elixir-compatible refactorings cataloged, with 32 (out of 82). Therefore, to improve the readability of this chapter, we chose to present in Table 4.6 and Table 4.7 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 C.1 (Appendix C).

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 [160]. In Elixir, pattern matching can be useful for assigning variables, unpacking values, and acting as a control-flow mechanism [6]. 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 [...]"*

Listing 4.4 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).

Listing 4.4: 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
```

This refactoring can be used to replace the `case` statement by a pattern matching over the parameter `n`, as shown in Listing 4.5. By performing this refactoring, `fibonacci/1`

Table 4.6: 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
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

#: Number of sources.

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 4.5: Example of a code after Introduce pattern matching over a parameter

```
1 # After refactoring:
2
3 def fibonacci(0), do: 0
4 def fibonacci(1), do: 1
5 def fibonacci(n) when is_integer(n) do
6   fibonacci(n-1) + fibonacci(n-2)
7 end
```

Table 4.7: Functional Refactorings compatible with Elixir - Part 2

Refactoring	**Description**	#
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
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.

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 4.6 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 4.6: 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 4.7 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 4.7: 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.[18] 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 4.7), therefore presents an opportunity for applying the refactoring

[18] `https://hexdocs.pm/elixir/comprehensions.html`

List comprehension simplifications, since a list comprehension is used to perform the first of the two transformation steps (line 4). Listing 4.8 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 4.10), 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 4.8: 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* [94]. 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 *anonymous functions* were originally implemented are updated to use the new *local function*. This refactoring is also known as *lambda lifting*, as described by the authors of the paper P6 [156]:

> *"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 4.9 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.

Listing 4.9: 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
```

As shown in Listing 4.10, 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 4.10: 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.* [32], is a code transformation of larger granularity, characterized by being a sequence of atomic 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 4.11 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 4.11: Example of a code before Convert nested conditionals to pipeline

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

```
12      {:error, :invalid_move}
13    end
14  end
```

Listing 4.12 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.

Listing 4.12: 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
```

> **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.

4.1.2.2 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 4.8.

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 [94]. Listing 4.13 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 4.13: 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 4.11, 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 4.12, this function can also be refactored using Pipeline using "with", as these two strategies can be applied in the same opportunities.

Listing 4.14 shows the result of refactoring `update_game_state/3` using Pipeline using "with". Using this transformation, the calls to `valid_move/2` (line 4), `players_turn/2` (line 5), and `play_turn/3` (line 6) were chained using pattern matching in each of the three clauses of a `with` statement, thus eliminating the need to use nested conditional statements. Additionally, unlike the transformation using Convert nested conditionals to pipeline (Listing 4.12), Pipeline using "with" does not require modifications in the interfaces of the functions involved in the transformation.

Listing 4.14: 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
```

Table 4.8: 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
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.

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 4.14, 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 4.15, 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 4.15: 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`[19] and `Agent`[20] are abstractions with specific purposes, `GenServer`[21] 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 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 4.16 presents another opportunity for using Generalise a process abstraction. `DatabaseServer` is a `Task` used beyond the suggested purpose of this abstraction. According to Jurić [94], 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 4.16, 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

[19] https://hexdocs.pm/elixir/Task.html
[20] https://hexdocs.pm/elixir/Agent.html
[21] https://hexdocs.pm/elixir/GenServer.html

performed by this `Task` is implemented in an infinite loop (line 7), thus this process never stops.

Listing 4.16: 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
```

The result of this refactoring is presented in Listing 4.17. 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.

Listing 4.17: Example of a code after Generalise a process abstraction

```
1  # After refactoring:
2
3  defmodule DatabaseServer do
4    use GenServer
5
6    def start_link() do
7      GenServer.start_link(__MODULE__, nil)
8    end
9
10   def get(server_pid, query_def) do
11     GenServer.call(server_pid,{:run_query, query_def})
12   end
13
14   @impl true
15   def handle_call({:run_query, query_def}, _, state) do
```

```
16      {:reply, run_query(query_def), state}
17    end
18  end
```

Since Elixir does not have an `elseif` statement, an alternative for this absence can be the use of nested `if..else` statements. However, this can be a verbose and not very maintainable solution, as shown by the `classify_bmi/2` function in Listing 4.18.

Listing 4.18: Example of a code before Transform nested "if" statements into a "cond"

```
1  # Before refactoring:
2
3  def classify_bmi(weight, height) do
4    {status, bmi} = calculate_bmi(weight, height)
5
6    if status == :ok do
7      if bmi < 18.5 do
8        "Underweight"
9      else
10       if bmi < 25.0 do
11         "Normal weight"
12       else
13         if bmi < 30.0 do
14           "Overweight"
15         else
16           "Obesity"
17         end
18       end
19     end
20   else
21     "Error in BMI calculation: #{bmi}"
22   end
23 end
```

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 since Elixir has a specific statement called `cond` that can be used as an alternative to a sequence of nested `if..else` statements. Listing 4.19 shows the refactored code, where a `cond` statement is used in the `classify_bmi/2` function to evaluate four different expressions (lines 8 to 11). This new implementation returns a `string` when the first expression with a result equal to `true` is found, thus maintaining the same behavior as the version before the refactoring.

Listing 4.19: Example of a code after Transform nested "if" statements into a "cond"

```
1  # After refactoring:
2
3  def classify_bmi(weight, height) do
4    {status, bmi} = calculate_bmi(weight, height)
5
6    if status == :ok do
7      cond do
8        bmi < 18.5 -> "Underweight"
```

```
9         bmi < 25.0 -> "Normal weight"
10        bmi < 30.0 -> "Overweight"
11        true       -> "Obesity"
12      end
13    else
14      "Error in BMI calculation: #{bmi}"
15    end
16  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 specific to this language (17.1% of the catalog).

4.1.2.3 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 4.9.

Despite being dynamically-typed languages, Erlang and Elixir offer `typespecs`,[22] 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[23] to perform static code analysis to detect type inconsistencies and potential bugs [94]. 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 [157]:

> *"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 4.20 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.

[22] `https://hexdocs.pm/elixir/typespecs.html`
[23] `https://www.erlang.org/doc/man/dialyzer.html`

Listing 4.20: Example of a code before Add type declarations and contracts

```
1 # Before refactoring:
2
3 @spec set_background({integer(), integer(), integer()}) :: atom()
4 def set_background(rgb) do
5   do_something()
6 end
```

Table 4.9: 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 calls to `functions` that have `modules`, names, and parameters 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.

Although the function `set_background/1` presented in Listing 4.20 is already documented using a `@spec` directive (line 3), this documentation can be improved by defining a custom data type, as shown in Listing 4.21. 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).

Listing 4.21: Example of a code after Add type declarations and contracts

```
1  # After refactoring:
2
3  @typedoc """
4      A tuple with three integer elements between 0..255
5    """
6  @type color() :: {red :: integer(), green :: integer(), blue :: integer()}
7
8  @spec set_background(color()) :: atom()
9  def set_background(rgb) do
10   do_something()
11 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* [94]. Listing 4.22 shows an example of using the `Kernel.apply/3` to dynamically call the function `Enum.sort/1`.

Listing 4.22: Example of a code before From meta to normal function application

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

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 4.23. 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 4.23: Example of a code after From meta to normal function application

```
1  # After refactoring:
2
3  Enum.sort([3, 4, 1, 2])
```

Another refactoring—Introduce processes—was found in five sources from our SLR and MSR studies. In P38 [106], 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 4.24 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.

Listing 4.24: 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
```

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 4.25 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.

Listing 4.25: 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.

4.1.2.4 Traditional refactorings

Overall, we cataloged 25 TRADITIONAL refactorings compatible with Elixir. Therefore, for the same reason stated in Subsection 4.1.2.1, we chose to present in Table 4.10 only those found in at least three sources. Due to this, eight refactorings from this category were not included in these tables; however, their descriptions can be found in Table C.2 in Appendix C of this work.

The refactoring found in the highest number of distinct sources was RENAME AN IDENTIFIER. As shown in Table 4.10, it was found in 73 sources from both our SLR, GLR, and MSR studies. According to Murphy-Hill *et al.* [131] and Golubev *et al.* [79], 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 4.1.

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 4.26 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

Table 4.10: 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.

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`.

Listing 4.26: 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
```

The refactored version of this code is presented in Listing 4.27. In this code, a new function `report/1` receives a `tuple` as a parameter (line 3) and lines 9 to 11 from Listing 4.26 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 4.27: Example of a code after Extract function

```
1  # After refactoring:
2
3  def report({company, contact, cancel_policy} = confirmation) do
4    IO.puts("Booking made at the company: #{company}")
5    IO.puts("Any doubt, contact: #{contact}")
6    IO.puts("For cancellations, see company policies: #{cancel_policy}")
7  end
8
9  def ticket_booking(passenger, air_line, date, credit_card, seat) do
10   check_availability(air_line, date)
11   |> documents_validation(passenger)
12   |> select_seat(seat)
13   |> payment(credit_card)
14   |> report()  # extracted function call!
15 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.

4.1.3 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 in determining the quality of results, in our SLR, we followed the procedures proposed by Kitchenham and Charters [99] to define relevant search terms, thus mitigating the risk of important papers for our research not being found or retrieving many articles out of context. For the same reason, we adhered to the guidelines proposed by Garousi *et al.* [76] and Dabic *et al.* [56] 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 4.2), 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 author and the advisor of this thesis 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. The author and the advisor of this thesis 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 advisor of this thesis, therefore minimizing individual perception biases. Hence, the decisions made in these steps reflected the consensus of the author and his advisor. Another threat to internal validity concerns possible missed refactorings in the artifacts analyzed in our MSR

study. As explained in step 10 of Section 4.1.1, 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. 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).

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. Moreover, in our MSR study, we selected only 10 repositories to investigate refactorings for Elixir. This is a small number since currently there are around 93K Elixir repositories on GitHub, so 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 4.2). In this survey, we did not receive any comments about missing refactorings.

4.2 Catalog Validation

In this section, we introduce the second study of this chapter, in which we surveyed Elixir developers to validate the catalog of refactorings outlined in Section 4.1. 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

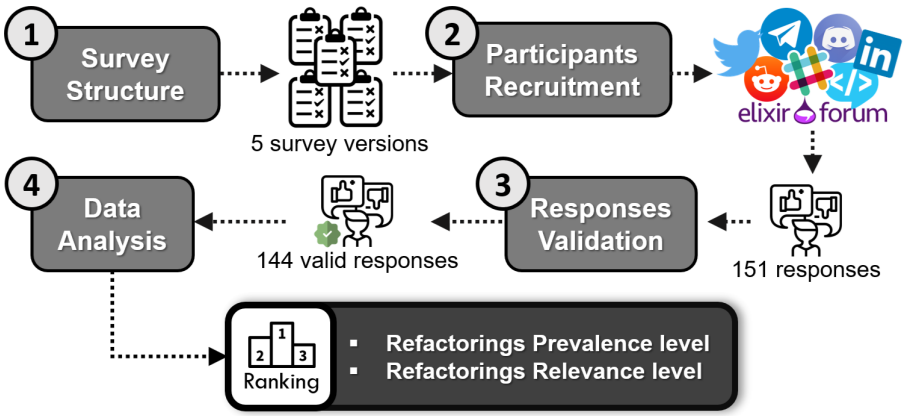
conducting the survey, it underwent evaluation and approval by the Research Ethics Committee at the Federal University of Minas Gerais, Brazil (CAAE: 76454823.8.0000.5149).

We dedicate Section 4.2.1 to outline the methodology employed in designing, conducting, and analyzing our survey results. In Section 4.2.2, we detail and discuss upon the findings regarding the prevalence and relevance of refactorings in Elixir. Finally, Section 4.2.3 addresses potential threats to the validity of our study and what measures we took to mitigate them.

4.2.1 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 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 4.3 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.

Figure 4.3: Overview of survey on refactorings for Elixir



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 (Appendix B.1). The questions posed to participants during this study are available in our replication package [201] and in Appendix B.2.

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 4.11.

Table 4.11: 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)

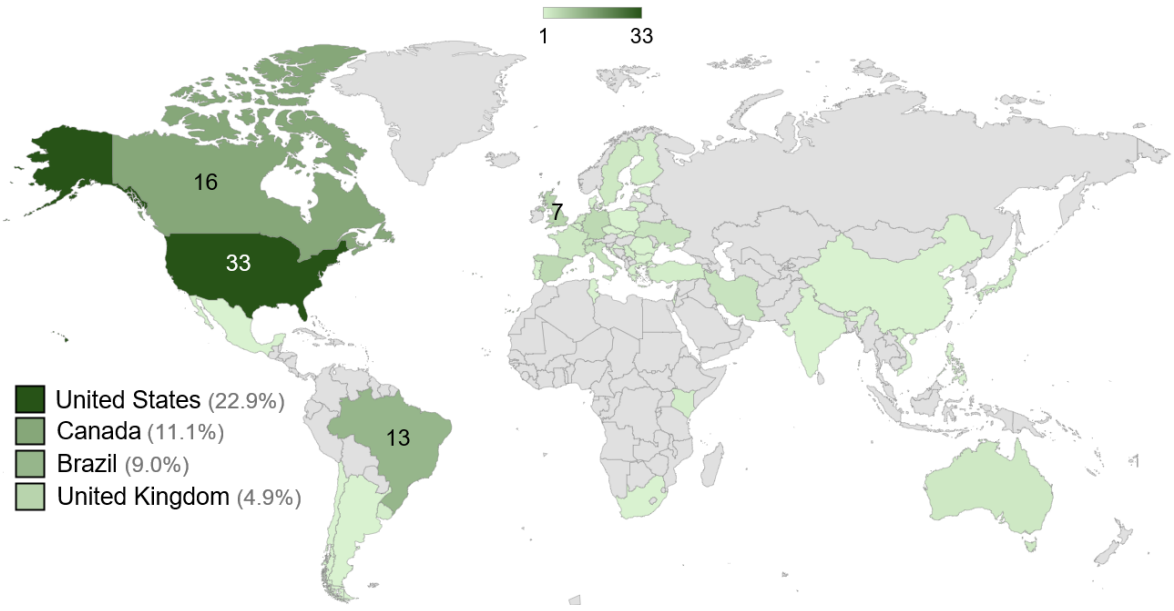
A distinct Google form was utilized to create each survey version. Moreover, we developed a simple script—available in our replication package [201]—that yields a random number between zero and four. Based on this number, the script also forwards the participants to the survey version they should answer to. This survey structure is similar to one used in our previous study on code smells for Elixir (Chapter 3).

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 (Chapter 3), 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,[24] 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,[25] Discord,[26] LinkedIn,[27] Reddit,[28] Slack,[29] and Telegram.[30]

We concluded the survey with 151 responses after two weeks. The recruited participants are spread across 42 countries spanning all continents. As depicted in Figure 4.4, 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.

Figure 4.4: Countries where the survey on refactorings participants reside



Note: The legend presents the top 4 countries with most participants.

[24] https://elixirforum.com/
[25] https://devtalk.com/elixir
[26] https://discord.gg/elixir
[27] https://www.linkedin.com/groups/6530248/
[28] https://www.reddit.com/r/elixir/
[29] https://elixir-slackin.herokuapp.com
[30] https://t.me/elixir_world

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, 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 4.12. 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.

Table 4.12: Valid responses for each survey version

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

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.

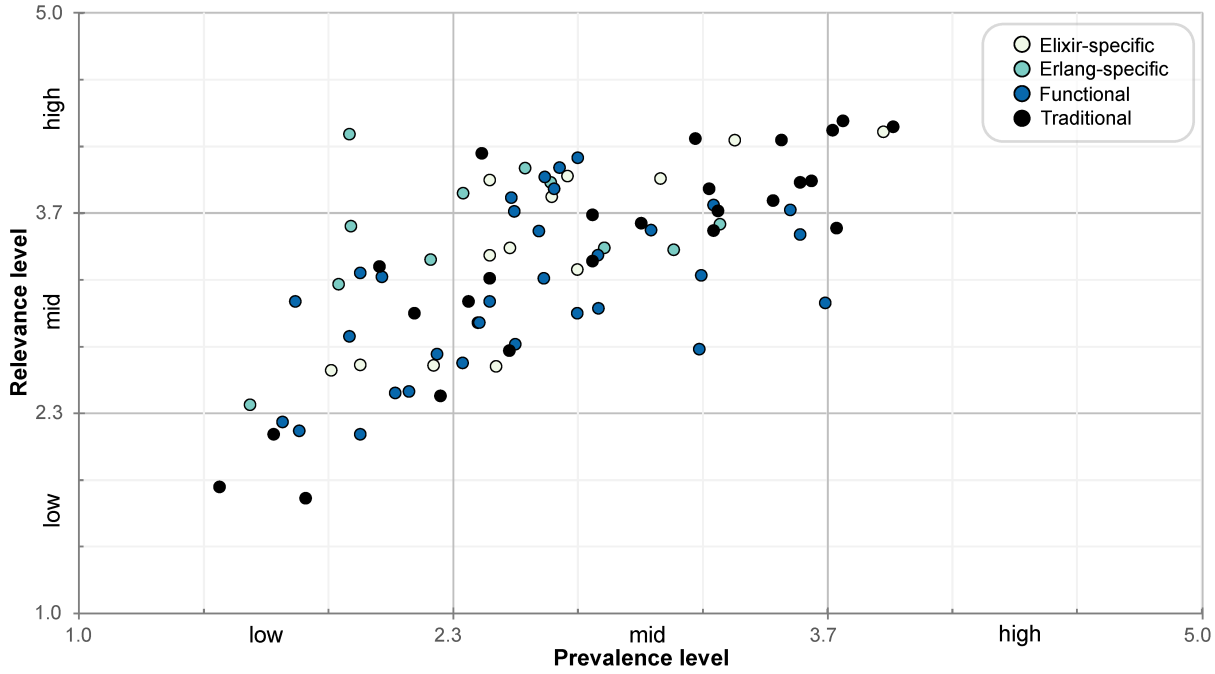
Finally, we employed the Mann-Whitney test [167] to evaluate whether the respondents' experience level affected their perception of the prevalence and relevance of refactorings. Due to the lack of a normal distribution in our data, we opted for this non-parametric test. The number of years a participant 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. We conducted these analyses using the SPSS statistical analysis tool,[31] with a significance level set at 0.05.

[31] https://www.ibm.com/spss

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

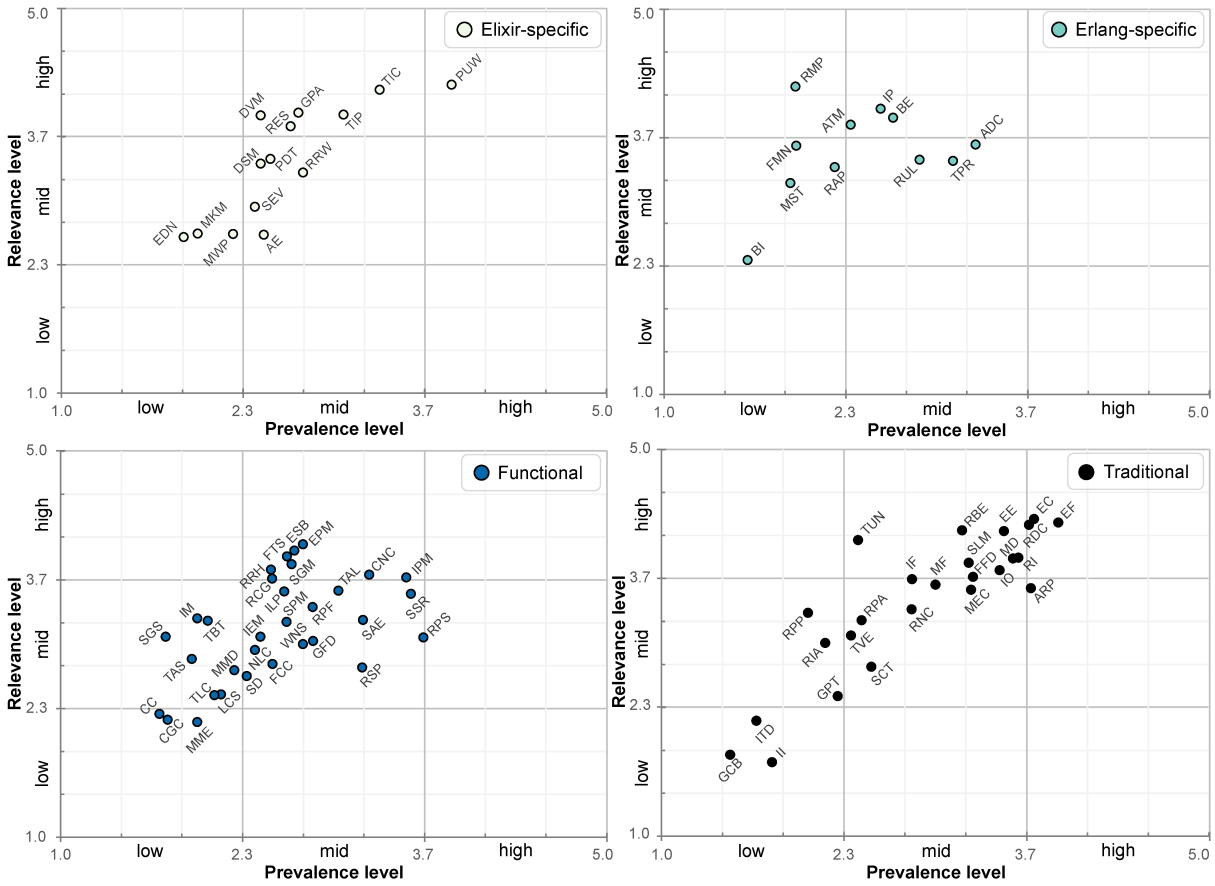
To facilitate the comparison between the refactorings for Elixir, we chose to present the results of our survey using scatter plots. Due to the large number of refactorings in our catalog, two different perspectives were used for this presentation. In Figure 4.5, we show a scatter plot containing all 82 refactorings from the catalog, thus providing an overview of the results. On the other hand, Figure 4.6 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).

Figure 4.5: Developers' perception of refactorings in Elixir (RQ2)



When analyzing each of the nine zones in Figure 4.5 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 Figure 4.6, 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 4.6: Developers' perception of refactorings in Elixir by category (RQ2)



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

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

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
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

Figure 4.5 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, Figure 4.6 shows that the three most prevalent refactorings are the traditional ones, including EXTRACT FUNCTION (EF, 3.90)[32] 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 Figure 4.6). 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 Figure 4.5, 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

[32]When discussing each refactoring, we are adding between parentheses the acronym used in Figure 4.6 (*e.g.,* EF) and the respective average score of the survey answers (*e.g.,* 3.90).

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 stood out the most in our catalog, we calculated the arithmetic mean of their relevance and prevalence levels. As shown in Table 4.13, nine (out of 82) refactorings have high-averages (*i.e.,* scores above 3.67)—seven traditional and two Elixir-specific—and, therefore, are those that deserve special attention from Elixir developers. Lastly, in order to also understand which categories stood out the most, we calculated the arithmetic mean of each one between its average relevance and average prevalence levels. As shown in Table 4.14, the category with the highest arithmetic mean was that of Traditional refactorings (3.15), closely followed by the Elixir-specific one (3.01). Not coincidentally, these are also the categories of the nine refactorings with high-averages.

Table 4.13: Refactorings with a high average 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.

Table 4.14: Averages 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

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.

Since the prevalence and relevance levels of each refactoring, shown in Figure 4.6, were calculated using the arithmetic mean of their respective responses, just as we did in Chapter 3, we decided to compute the standard deviation (SD) for both prevalence and relevance. The SDs for prevalence range from 0.72 to 1.82, with Behaviour inlining (BI) showing the lowest variation of the responses relative to the mean and Remove unnecessary calls to length/1 (RUL) demonstrating the most heterogeneous responses. For relevance, the SDs range between 0.82 and 1.68, where Transform "if" using pattern matching into "case" (TIP) stands out with the most homogeneous responses, while Alias expansion (AE) exhibits the highest variation. Detailed values for these standard deviations are provided in the replication package of this chapter [201].

The Mann-Whitney test results indicate that the developers' experience level affected how they perceived just 16 refactorings from our catalog. Table 4.15 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 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 4.15 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 4.16 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' per-

Table 4.15: Influence of the Elixir experience in the developer's perception of a refactoring

Refactoring (Perception)	**Mean rank**		*Sig.*
	≤ 3 years	**> 3 years**	
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.

ception of the prevalence than of the relevance of these code transformations. Out of the 18 perceptions[33] 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.

Table 4.16: Influence of the number of Elixir projects in the developer's perception of a refactoring

Refactoring (Perception)	**Mean rank**		*Sig.*
	≤ 4 projects	**> 4 projects**	
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.

[33] As can be seen in Table 4.15 and Table 4.16, 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.

> **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 of this chapter [201] 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.[34]

4.2.3 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 (Chapter 3). 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 [204]. To mitigate this threat, we normalized the answers for each of our five questionnaire versions, thereby compensating for the

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

imbalance. Another threat 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.

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 author of this work was available via email to clarify any doubts respondents had regarding refactorings and the questions to be answered. 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.* [133] 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 [167] 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.

4.3 Implications

Based on the results presented in this chapter, 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 [215]. 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[35] 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 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.

4.4 Final Remarks

This chapter 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 prospect and document refactorings. Specifically, 391 research papers, 111 grey literature documents, and 488 artifacts mined from the Top-10 Elixir repositories with the most stars on GitHub were analyzed to catalog 82 refactorings that can be applied in Elixir systems.

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.

Only six of the 82 refactorings cataloged in this thesis—Generalise a process abstraction, Moving error-handling mechanisms to supervision trees, Introduce processes, Remove processes, Add a tag to messages, and Register a process—directly involve concurrent programming features. Like the other 76 sequential refactorings, these **concurrent refactorings preserve the original behavior of the transformed code due to the BEAM VM's design, which inherently provides thread safety**. Specifically, concurrent code creates multiple processes on the BEAM [94]. These processes operate independently, sharing no memory and communicating exclusively through message passing [183]. Moreover, each process handles a single request at a time, maintaining consistency and thereby avoiding common concurrency issues such as race conditions [94]. Thus, **the process isolation model provided by**

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

the BEAM guarantees behavioral preservation in concurrent refactorings, even without the explicit use of synchronization mechanisms required in other languages.

It is important to acknowledge that we do not claim completeness for the proposed catalog. Although we have sought to provide a comprehensive catalog, we believe it is possible to extend this catalog through different methodologies for identifying refactorings or even by applying the same methods used in this study at a future time, when new discussions may have emerged within the developer community or when new scientific papers on refactorings for functional languages may have been published.

We summarize the contributions of this chapter as follows:

- We cataloged 82 refactorings for Elixir and categorized them into four different groups, Elixir-Specific refactorings (14), Functional refactorings (32), Erlang-Specific refactorings (11), and Traditional refactorings (25).
- We provided documentation with code examples and some tailored side conditions that can support the implementation of automated refactoring tools for Elixir in the future. Indeed, no robust, up-to-date, and widely adopted refactoring tool is available for Elixir, as shown by an exploratory search conducted by us on Hex,[36] Elixir's package manager.
- We showed that most of the cataloged refactorings are at least moderately prevalent, indicating they are common in production code.
- Furthermore, we have shown that the vast majority of refactorings are at least moderately relevant, suggesting they have the potential to enhance the quality of systems developed in Elixir.
- Moreover, we found that nine refactorings in the catalog have a high average score between their relevance and prevalence levels, indicating they deserve special attention from Elixir developers.
- Finally, we found that the experience level of the Elixir developers had little impact on their perceptions of the relevance and prevalence of the refactorings in our catalog.

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.

Replication Package. We provide the complete dataset used in this chapter and a replication package at: `https://doi.org/10.5281/zenodo.11372758`.

[36] `https://hex.pm/`

Chapter 5

Relationship between Code Smells and Refactorings in Elixir

In this chapter, **we propose a mapping between Elixir code smells (Chapter 3) and the refactorings cataloged in this thesis (Chapter 4)**. In total, 176 relationships were mapped between all 35 code smells and 70 corresponding refactorings that can be useful in transformations that eliminate them. Additionally, **we were able to identify five new composite refactorings for Elixir during this mapping process**, which were not cataloged in Chapter 4. To accomplish this, we conducted an empirical study where each of the 35 code smells proposed in this thesis (Chapter 3) was manually compared with each of the 82 refactorings cataloged by us (Chapter 4). Through these comparisons, we identified the refactorings that could aid in removing each smell, which ones they are, and in what order they should be performed.

The methods used in this chapter to establish and describe relationships between code smells and refactorings is similar to the one used by Fowler and Beck [74] to correlate their well-known catalogs and guide developers when they are unsure on how to improve the quality of their sub-optimal code structures. Many other authors have also conducted studies that empirically mapped refactoring strategies to eliminate code smells [10, 34, 68, 107, 108, 113, 162, 188].

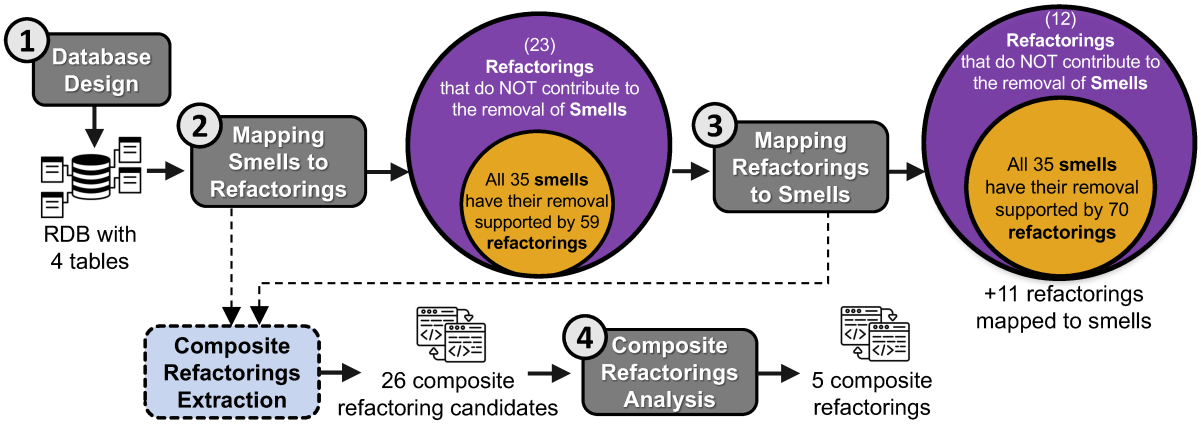
This chapter is organized as follows. In Section 5.1, we detail the methods used to correlate code smells with refactorings that can assist in their removal. Next, in Section 5.2, we present the results of this mapping between smells and refactorings, detailing step-by-step how some of these code transformation strategies can be used in transformations to remove sub-optimal structures. Additionally, in this section, we present five new composite refactorings useful in removing code smells in Elixir. In Section 5.3, we discuss the practical implications of our findings and also possible reasons that justify the absence of relationships for 12 (out of 82) refactorings in our catalog. Potential threats to validity and ways to mitigate them are presented in Section 5.4. Finally, we conclude this chapter in Section 5.5.

5.1 Study Design

According to Fowler and Beck [74], while it is important to catalog the mechanics of a refactoring strategy, it is also essential to document when each of these code transformations should be applied. Complementarily, according to Sharma *et al.* [164], merely cataloging the quality issues caused by each code smell is not sufficient for their disciplined removal. For this reason, unveiling which refactorings are appropriate for the removal of these smells is equally important.

Considering the importance of establishing this type of relationship between smells and refactorings, and that this Ph.D. thesis takes inspiration from Fowler's book [74] to promote quality improvements in systems developed in Elixir, we decided to map the relationships between code smells and refactorings in this language in a manner similar to that carried out by Fowler and Beck [74], thus maintaining coherence between our work and one of its main inspirations. Figure 5.1 summarizes the steps we followed to correlate code smells and refactorings for Elixir. We also detail all these steps in the following paragraphs.

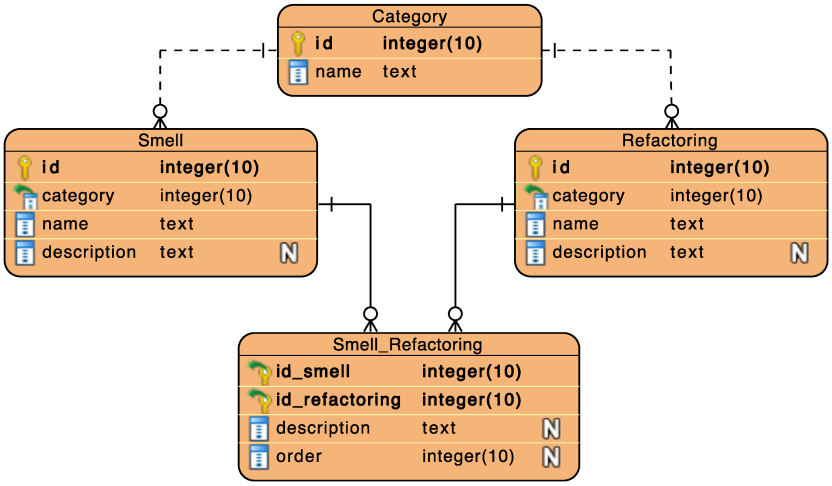
Figure 5.1: Overview of methods for correlating code smells and refactorings in Elixir



1) Database Design: As observed in the mappings between code smells and refactorings conducted by Fowler [74] and also by Ferreira *et al.* [68], the same refactoring strategy can be used in the removal of different types of code smells. Additionally, a code smell can also be removed with the contribution of different refactoring strategies, which may vary according to the problem faced by the developer. Considering this recurring behavior in the literature, the potential for the mapping investigated in this chapter to include a large number of relationships was significant, given that we have cataloged 35 code smells and 82 refactorings for Elixir.

To handle this issue, we opted to structure a relational database (RDB) using the SQLite[1] embedded database engine and then use it to persist all the relationships found in the mapping between smells and refactorings. Figure 5.2 presents the database schema modeled to persist these data. The relationship between the `Smell` and `Refactoring` tables was designed as many-to-many, thus generating the `Smell_Refactoring` table, where all the data found in the subsequent steps of this study were persisted.

Figure 5.2: Database used to document treatments to code smells by refactoring strategies



According to Bibiano *et al.* [24, 25] and Cedrim *et al.* [43], refactorings are considered incomplete when they fail to remove a code smell completely. Considering that a refactoring might not always have the capability to completely remove an instance of a code smell on its own, we added the `order` column to the `Smell_Refactoring` table. This column is intended to define the position of a refactoring within a sequence of interrelated refactorings aimed at removing the same code smell. In other words, the `order` column allows us to identify composite refactorings [32, 178] for removing code smells in Elixir. When a refactoring is not used in conjunction with other transformation strategies to aid the removal of a smell, this column has a null value persisted in the respective relationship.

The resulting database from the study conducted in this chapter, along with a set of useful SQL queries, is available online in the replication package of this chapter.[2]

2) Mapping Smells to Refactorings: In this step, the author of this work manually compared each of the 35 code smells for Elixir with the 82 refactorings cataloged for this language. Essentially, the main objective of this step was to establish relationships between code smells and the refactoring strategies capable of contributing to their removal. To achieve this, the characteristics of the problems caused by the code smells were compared with the code each of the Elixir refactorings can generate. When it was identified that a problem caused by a sub-optimal structure is part of the motivation for perform-

[1] `https://www.sqlite.org/`

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

ing a particular refactoring, which would thus remove it at least partially, a relationship between a code smell and a refactoring was mapped.

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

Parallel to this step, the activity *Composite Refactoring Extraction* was also conducted, as shown in Figure 5.1. In this parallel activity, whenever a relationship between a code smell and a refactoring was identified, the author of this work empirically analyzed whether the refactoring in question is sufficient to remove the mapped code smell on its own, or if this code transformation needed to be complemented by other refactorings performed in a specific sequence to enhance the removal. This parallel activity therefore allowed us to identify composite refactoring candidates [32, 178] for Elixir, something not directly explored in Chapter 4.

3) Mapping Refactorings to Smells: Considering the manual and subjective nature of the mapping carried out in the previous step, and also that a code smell can eventually be removed in different ways depending on the specific problem a developer has at hand [68, 74], in this third methodological step, the author of this work manually compared each of the 23 refactorings not yet associated with any code smell in the previous step with all 35 code smells cataloged for Elixir. The objective of following the reverse path of the previous step and additionally using only a specific subset of the refactorings was to mitigate any potential flaws in the previous mapping process that may have caused the absence of relationships for these 23 refactorings.

The aspects compared between the code smells and the refactorings in this step were the same as those described in the step 2, with the main difference being that the initial comparative reference here was the refactorings, not the code smells as previously. Considering the sets of refactorings and code smells compared in this third step, we had the potential to identify 805 relationships between them, representing the Cartesian product of the involved elements. Out of these, 12 new relationships were mapped, involving 11 refactorings and seven smells. Among these refactorings involved in removing code smells, only Remove import attributes was mapped in two different removals. The remaining 10 were each mapped to the removal of only one code smell. Regarding the code smells involved in these new relationships, one was mapped four times (Long Function) and another three times (Duplicated code). The remaining five were each mapped only once. Therefore, at the end of this step, the number of refactoring strategies not associated with the removal of cataloged code smells decreased to 12. In Section 5.3, we analyze and

discuss possible reasons that justify the absence of relationships for these refactorings.

As in the previous step, the *Composite Refactoring Extraction* activity was also conducted parallel to this third step. Thus, after completing the mappings between the code smells and refactorings for Elixir, in addition to identifying how each code smell can be systematically removed with the help of atomic refactoring operations, the author of this work also identified 26 different interrelated sequences of refactorings that can be used to assist the removal of code smells in Elixir. The resulting mapping and the identified composite refactoring candidates were discussed with the advisor of this thesis, who agreed with and validated the decisions made by the author.

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

The objective of this fourth and final methodological step was to quantify the number of times each of the 26 composite refactoring candidates identified in the previous steps was mapped and then give a name to only those that were recurrent. In total, five of these 26 complex transformations were mapped more than once and were therefore considered composite refactorings for Elixir. Out of these five recurrent composite refactorings, four resemble other complex code transformations previously discussed in the literature in different contexts [32, 74, 86]. Therefore, we used the names of these refactorings as originally proposed by other authors as inspiration for naming these composite refactorings for Elixir.

5.2 Results

In Subsection 5.2.1, we present the mappings found between Elixir smells and the corresponding refactorings that can be useful in transformations performed to eliminate them. Moreover, we selected a subset of four code smells to provide a more detailed explanation of how their elimination can be aided by refactoring strategies. These smells were selected for being the most relevant or prevalent in Elixir (Chapter 3), making their removal important and representative for developers working with this language. In Subsection 5.2.2, we present five new composite refactorings that contribute to the removal of code smells in Elixir. These composite refactorings were identified during the mappings performed in this study. Finally, in Subsection 5.2.3, we present a complete example of removing a code smell step-by-step using a composite refactoring in Elixir.

For this, all intermediate versions between the original code and the fully refactored code are presented.

5.2.1 Mapping between smells and refactorings

In our mapping study, we found that all 35 code smells are covered by at least one of the refactorings in our catalog, meaning there is at least one refactoring that helps in the removal of these smells. In total, 176 relationships between code smells and refactorings for Elixir were identified, demonstrating that a code smell can have their removal aided by more than one distinct refactoring strategy, and also that a refactoring can be useful in the removal of multiple code smells. Due to the large number of relationships between smells and refactorings found in this study, to improve the readability of this chapter, we chose to present in Table 5.1 and Table 5.2 only the smells that have their removal aided by no more than five different refactorings. Consequently, the mappings for the remaining 10 smells that do not meet this criterion can be found in Appendix D.

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

Dynamic atom creation is Elixir's second most relevant smell. This smell occurs when a function creates `atoms` in an uncontrolled and dynamic way. Since values of this Elixir basic type are not garbage collected by BEAM, this lack of control by the developer over how many `atoms` will be created during an application's execution cycle can expose the software to unexpected behaviors caused by excessive memory usage. We can contribute to the removal of this smell by replacing calls to the `String.to_atom/1` function, which dynamically creates `atoms`, with explicit conversions. To do this, we should use the

Table 5.1: Elixir smells and the refactorings that assist their elimination - Part 1

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

Table 5.2: Elixir smells and the refactorings that assist their elimination - Part 2

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

Extract function refactoring on the calls to `String.to_atom/1` to create a new function. This new function should accept a string as a parameter and convert it to a statically predefined `atom`. The body of this new function should include a conditional that checks the

content of the string. Depending on the string's content, the function will return a different predefined `atom` directly, without using `String.to_atom/1`. After extracting a new function for explicit conversions from strings to atoms, we can also use the refactoring Introduce pattern matching over a parameter to transform the extracted function into a multi-clause function, where each clause is responsible for returning one of the possible statically predefined atoms.

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

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

Finally, Working with invalid data is the second most prevalent smell for Elixir according to developers. This smell occurs when a function does not validate its parameters and propagates them to other functions, which can lead to unexpected internal behavior. When a library function does not validate the types of its parameters, we can at least create a *Proxy* [75] function for this smelly library function and use the Typing parameters and return values refactoring to document the types of these data directly in the proxy. This will help clients of the smelly function (*i.e.,* third-party code) protect themselves from potential errors caused by invalid data. If recurring data structures are identified while documenting a function using Typing parameters and return values, these structures can be named using the Add type declarations and contracts refactoring. This approach creates new reusable data types and enhances the system's readability.

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

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

5.2.2 Composite refactorings for Elixir

We found five recurring sequences of complementary atomic refactorings that can be useful to remove code smells in Elixir. In other words, in this study, we also cataloged five new composite refactorings for Elixir, as presented in Table 5.3.

Table 5.3: Composite refactorings for Elixir

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

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

Extract to outside is a sequence of atomic refactoring operations that contributes to the removal of five different code smells in Elixir, making it the most recurrent composite refactoring among those cataloged in this study. This refactoring is characterized by being a sequence composed of an Extract function, followed by a Moving a definition. Generally, this sequence can be used to break down an original function

into smaller parts and move them to modules that group other functions with similar objectives. Therefore, this composite refactoring aims to improve the understandability of a function while maintaining the cohesion of the module where it is originally defined. The mechanics of Extract to outside are analogous to Method decomposition [32], which is an equivalent composite refactoring cataloged for object-oriented code.

The second most recurrent composite refactoring for Elixir is Module decomposition. Its name is similar to another equivalent composite refactoring—Class decomposition—cataloged by Brito *et al.* [32]. Some instances of code smells in Elixir are characterized by modules with too much behavior or modules that collaborate excessively and are too highly coupled. This refactoring can address smells with these characteristics by improving the modularity and cohesion of a system, while also reducing unnecessary coupling between modules. To achieve this, the refactoring employs a sequence composed of Splitting a large module, followed by Rename an identifier, as breaking a module into smaller ones may require updating the name of the original module to better reflect its new purpose.

Introduce parameter struct is analogous to the traditional refactoring Introduce parameter object, cataloged by Fowler [74]. Although this traditional refactoring is not composite, both aim to group related data into a reusable data structure. However, to achieve a result equivalent to Introduce parameter object with Introduce parameter struct, we need to perform two distinct atomic refactorings sequentially: Grouping parameters in tuple and From tuple to struct, respectively.

Gradual change is a composite refactoring for Elixir that can occur over several different commits until its completion. It involves code transformations that coexist with the original versions for a certain period, thereby avoiding breaking changes. The expressions or modules involved in this composite refactoring are initially duplicated using the atomic refactoring Introduce a temporary duplicate definition. The original versions are set as deprecated while the actual changes are applied to the duplicates. After a period with the original versions deprecated, they are removed using the atomic refactoring Remove dead code. The purpose of Gradual change is similar to the technique *Branch by Abstraction* [86], commonly used in continuous delivery processes to implement large-scale changes to a software system gradually, allowing for regular releases even while the change is still ongoing.

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

As we can see in Table 5.3, two of the five composite refactorings cataloged for Elixir in this study—Introduce parameter struct and Gradual change—have the capability to aid the elimination of the smells they were mapped to without requiring any additional refactorings. The other three composite refactorings sometimes need to be used in conjunction with other refactorings to enhance the removal of the mapped smells. For example, instances of the Large code generation by macros smell can be completely removed using just the composite refactoring Extract to outside. Similarly, instances of the Divergent change smell can be removed using Module decomposition. However, during the elimination of Data manipulation by Migration instances, we need to apply the following sequence of refactorings: Module decomposition → Extract to outside → Remove dead code.[3]

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

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

5.2.3 Example: Removing a smell step-by-step through a composite refactoring

In this section, the smell Large code generation by macros is removed step-by-step using the composite refactoring Extract to outside to illustrate how this type of refactoring can be applied in Elixir. As shown in Table 5.3, this composite refactoring is characterized by the following sequence of atomic refactorings: Extract function → Moving a definition.

Macros are meta-programming mechanisms in Elixir that can extend the language.[4] They enable robust code generation at compile time, which helps reduce boilerplate and allows the creation of DSL constructs for Elixir [94]. The code smell used as an example in this section is related to macros that generate too much code. When a macro generates a large amount of code, it impacts how the compiler or the runtime work. The reason

[3]This sequence is a composite refactoring *candidate* for Elixir. However, since it does not recur in removing other code smells like the five listed sequences in Table 5.3, it was not named.

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

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

Listing 5.1 shows a module used to access a router for a web application. The function `show/0` (line 7) lists all the routes defined for this application. They are stored in the Elixir's module attribute `@store_routes`, which is created and modified by the calls to the macro `Routes.get/2` (lines 4 and 5).

Listing 5.1: Router for a web application

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

As shown in Listing 5.2, the macro `get/2` (lines 3 to 19) is an instance of the Large code generation by macros smell. On every invocation of this macro, which could be hundreds, the code inside `get/2` is expanded and compiled, which can generate a large volume of code overall. This occurs because most of the code defined in `get/2` could be implemented in a conventional function, thus avoiding excessive expansions and compilations.

Listing 5.2: Instance of the smell Large code generation by macros

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

The first step in removing this smell is to perform Extract function. With this refactoring, we can extract part of the macro's code and encapsulate it into a conventional function, which the macro will then call. As shown in Listing 5.3, by extracting to the function `__define__/3` (line 9) the code originally defined inside the `quote/1` (Listing 5.2 - lines 4 to 18), we reduce the amount of code that is expanded and compiled on every invocation of `get/2`, and instead we dispatch to `__define__/3` to do the bulk of the work.

Listing 5.3: Intermediate code version after Extract function

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

After extracting the function `__define__/3` into the `Routes` module (Listing 5.3), we can also move it to the `Routes.Utils` module using the Moving a definition refactoring, thus making the code more cohesive (Listing 5.4). This occurs because, although omitted in Listing 5.4, `Routes.Utils` groups other functions with the same objective as `__define__/3`. As a result of this second step, the code smell Large code generation by macros is completely removed, since the macro now generates only the minimum necessary amount of code to maintain the original system behavior, and we also achieve cleaner and more organized code.

Listing 5.4: Code smell is completely removed after using a composite refactoring

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

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

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

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

5.3 Discussion

Since 12 of the 82 refactorings cataloged for Elixir were not mapped to removing any code smell, we sought to understand the reasons for these mapping absences. To do this, we used a taxonomy proposed by Abid *et al.* [1] to classify the main motivations behind performing a refactoring strategy. After the author of this thesis classified the motivations behind each of the 12 code transformation strategies not mapped to the removal of smells, the advisor of this thesis validated these classifications, which are presented in Table 5.4.

Table 5.4: Refactorings not mapped to the removal of code smells

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

According to the taxonomy used in this classification, the motivation for improving internal quality involves, for example, maintainability, flexibility, portability, reusability, or readability code issues. On the other hand, the motivation for improving performance involves aspects such as response time, error rate, request rate, memory use, or code parallelization. In addition to these two motivations, the taxonomy proposed by Abid *et al.* [1] includes three other categories which clearly do not have any relation with the 12 unmapped refactorings: external quality, security, and migration issues.

As shown in Table 5.4, five of the 12 unmapped refactorings are motivated by performance concerns. This explains their absence in removing code smells in Elixir, as these sub-optimal code structures are more related to aspects that hinder the internal quality of systems [74]. On the other hand, the other seven unmapped Elixir refactorings focus on improving internal quality, which at first could suggest the existence of uncataloged code smells for Elixir. However, six out of these seven refactorings primarily address minor coding style adjustments, such as replacing certain conditional statements or substituting calls to Elixir's built-in functions with specific operators of this language. Given that code smells generally involve structures with more substantial granularities, this explains the absence of mappings for these six refactorings. The lack of mappings for Replacing recursion with a higher-level construct, however, might indeed indicate the existence of an uncataloged code smell. This refactoring involves significant code block transformations, potentially impacting the design of functions or modules. This type of change is compatible, for example, with the removal of an uncatalogued Design-related smell (Chapter 3).

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

> **Finding #3:** We have found evidence suggesting the existence of one uncatalogued Design-related smell for Elixir.

Regarding the unmapped refactorings, as shown in Table 5.5, the Traditional

refactorings category had the lowest proportion of refactoring strategies not associated with code smell removal in Elixir (8.00%). Furthermore, 51.14% of the 176 relationships mapped in this study involved TRADITIONAL refactorings, making it the category that contributes the most to code smell removal. These data show that refactorings cataloged 25 years ago by Fowler and Beck [74], although originally proposed for a different context, remain highly useful and relevant even for a specific scenario like a functional language such as Elixir.

Table 5.5: Overview of refactorings by category (Unmapped x Mapped)

Category	**# Refactorings**	**# Unmapped (%)**	**# Relationships (%)**
TRADITIONAL	25	***2 (8.00)***	***90 (51.14)***
FUNCTIONAL	32	7 (21.88)	46 (26.14)
ERLANG-SPECIFIC	11	1 (9.09)	21 (11.93)
ELIXIR-SPECIFIC	14	2 (14.29)	19 (10.80)

> **Finding #4:** TRADITIONAL refactorings proposed to improve the quality of object-oriented systems are also highly important for removing code smells in Elixir.

5.4 Threats to Validity

Construct Validity: Considering that a single refactoring might not always be sufficient to fully eliminate a code smell, a threat to the construct validity of this study is that our mapping might lead developers to perform incomplete refactorings [24, 25, 43] due to a lack of knowledge about the necessary complementary steps to completely remove a smell, or at least address the largest possible portion of it. To mitigate this threat, we not only mapped simple relationships between smells and atomic refactorings, but also documented 26 composite refactoring candidates and five composite refactorings for Elixir. These sequences of code transformations used together can help developers reduce the occurrence of incomplete refactorings in Elixir codebases, thereby promoting smell removals that are more aligned with real-world needs.

Conclusion Validity: The primary concern regarding this type of threat in our mapping study relates to potential biases in qualitative analyses that could compromise the reliability of the work. Since all the initial mappings between smells and refactorings, as well as the identification of composite refactoring candidates, were conducted solely by the author of this study, these analyses might have been influenced by personal experiences and

perspectives, potentially compromising the results. To address this threat, the resulting mappings and the identified composite refactoring candidates were also discussed with the advisor of this thesis, who reviewed, agreed with, and validated the decisions made by the author.

Internal Validity: The main threat to the internal validity of this study concerns the possible existence of relationships between smells and refactorings not captured by the methodological steps employed, which could influence the quality of our findings. Given the manual and subjective nature of the mapping process carried out in this study, it is natural to consider that it is susceptible to human error in identifying these relationships. To mitigate this risk, in addition to conducting an initial comparison between all 35 smells and 82 refactorings for Elixir (Section 5.1 - step 2), we also implemented a second comparison activity (Section 5.1 - step 3) aimed at finding relationships for the 23 refactorings not associated with any code smell in the previous step. This additional step helped to reduce potential flaws in this manual comparison process, as it allowed us to find relationships for 11 of the 23 refactorings initially not correlated to the removal of code smells. Additionally, we used the taxonomy proposed by Abid *et al.* [1] to classify the motivations behind the 12 refactorings not associated with code smells and concluded that this absence of relationships is not related to the methods used in this study.

External Validity: This threat concerns the generalization of the relationships found between the catalogs of code smells and refactorings, since these smells may not represent all possible quality issues in Elixir systems, and these refactorings might not be the only ways to address these sub-optimal structures. Despite this risk, both catalogs compared in this study have been extensively validated with experienced developers in previous studies. Additionally, the mappings between both catalogs include refactorings that are useful for removing all cataloged smells, and among the refactorings not associated with smell removal, only one shows indications of an uncataloged smell for Elixir. This suggests the robustness and high level of completeness of our results, which mitigates this threat.

5.5 Final Remarks

This chapter proposes a mapping between the code smells (Chapter 3) and the refactorings (Chapter 4) cataloged for Elixir in this thesis, indicating which refactorings may be useful in code transformations carried out to remove each code smell. To establish these relationships between the two catalogs, we manually compared the characteristics of the problems caused by each code smell with the motivations and the code improvements

each of the Elixir refactorings can generate.

We summarize the contributions of this chapter as follows:

- We found that all 35 code smells for Elixir are covered by at least one of the refactorings in our catalog, meaning there is at least one refactoring that helps in the removal of these smells.
- We showed that some refactoring operations cataloged for Elixir can be useful for addressing more than one code smell, thereby highlighting their versatility in solving these issues.
- On the other hand, we found that 12 of the 82 refactorings cataloged for Elixir are not associated with the removal of known code smells for this language.
- We identified five new composite refactorings that can be useful in removing code smells in Elixir. Three of them can be used in conjunction with other refactorings to assist the removal of the smells they were mapped to.
- We have found evidence suggesting the existence of an uncatalogued Design-related smell for Elixir.
- Finally, we found that the traditional refactorings proposed by Fowler and Beck [74] to improve the quality of object-oriented systems are also highly important for removing code smells in Elixir.

These findings have practical implications. For example, the mapping between catalogs conducted in this study can guide developers, especially those beginners to Elixir, on how to systematically remove code smells and improve the internal quality of their systems implemented with this language. Additionally, this guide can serve as inspiration for removing code smells in systems implemented in other functional languages. Finally, our research indicates the need for an investigation to validate with Elixir developers the prevalence and relevance of the new code smell identified through the analysis of unmapped refactorings. It also points to the need for further research into the existence of other Elixir-specific composite refactorings, as although we identified five of them in this study, this was not the primary focus of our investigation.

Replication Package. We provide the complete dataset used in this chapter and a replication package at: `https://doi.org/10.5281/zenodo.13835771`.

Chapter 6

Conclusion

This chapter concludes the thesis by discussing the main contributions and suggesting directions for future work. In Section 6.1, we provide an overview of the thesis. Section 6.2 summarizes the results and highlights the major contributions of this research, while Section 6.3 explores potential topics for future work.

6.1 Thesis Recapitulation

In the late 1990s, Fowler published his well-known book focused on promoting the disciplined improvement of existing code [74]. This book comprises two catalogs: one containing 22 code smells, which are sub-optimal code structures, and another featuring 72 code transformation strategies, known as refactorings, which can be used to eliminate code smells. Reflecting the popularity of this book, numerous studies have been conducted in the years following its publication up to the present day, aiming to understand various aspects related to code smells and refactorings. However, much like Fowler's book [74], these works have primarily focused on object-oriented programming languages [1, 176].

Historically, functional languages have not been as prevalent in the industry as object-oriented ones. However, interest in functional languages has recently risen in this environment [26]. More specifically, Elixir is a modern functional programming language that is gaining traction in the industry, with over 300 companies worldwide using it. In contrast to this recent popularity, to the best of our knowledge, no study has yet investigated code smells or refactorings specific to Elixir. In this context, we took advantage of this research opportunity and, in this Ph.D. thesis, conducted a set of three major studies in which we cataloged code smells and refactorings specifically for Elixir.

In Chapter 2, we provide an overview of the functional programming paradigm and the main characteristics of the syntax and semantics of the Elixir functional language. Additionally, we discuss the extensive literature on code smells and refactorings, as well as outline potential research opportunities related to these topics. Finally, we

compare the studies conducted in this thesis with related work investigating code smells and refactorings in other specific contexts, including some functional languages such as Haskell and Erlang.

Next, we reported in Chapter 3 our first study, where we cataloged and validated 35 code smells for Elixir. To identify and catalog these smells, we employed a mixed methodology approach based on a grey literature review [76], direct interaction with developers who work with Elixir, and mining code repositories on GitHub. We also surveyed 182 developers from 37 countries across all continents to measure the relevance and prevalence of each code smell, thus validating the proposed catalog.

In Chapter 4, we conducted a study structurally similar to the one reported in Chapter 3, which resulted in the cataloging of 82 refactorings for Elixir. The methodology used to catalog these refactorings was based on a systematic literature review [99], followed again by a grey literature review and mining code repositories on GitHub. The validation of these code transformation strategies was carried out through a second survey, where 151 experienced Elixir developers from 42 countries ranked the relevance and prevalence of the cataloged refactorings. Both surveys (Chapters 3 and 4) were previously approved by the Research Ethics Committee at the Federal University of Minas Gerais.

Finally, in Chapter 5, we reported an empirical study in which each of the 35 code smells proposed in this thesis (Chapter 3) was manually compared with each of the 82 refactorings we cataloged (Chapter 4). Through these comparisons, we identified which refactorings could help remove each smell and in what order they should be performed. A key contribution of this study was the proposal of practical guidelines for systematically removing code smells in Elixir systems, using refactoring strategies specific to the language (Appendix D). This guide follows a format similar to the one used by Fowler in his book to correlate code smells and refactorings for object-oriented systems [74].

6.2 Contributions

We summarize our contributions as follows:

- We initially proposed a **comprehensive catalog of 35 code smells for Elixir** (Chapter 3). This catalog includes 23 novel Elixir-specific code smells and 12 traditional ones (as proposed by Fowler and Beck [74]) that also occur in Elixir systems. We categorized the Elixir-specific smells into two groups: nine Low-Level Concerns smells, which have a narrow scope and affect small code structures, and 14 Design-Related smells, which are more complex and related to code orga-

nization, thus impacting larger portions of code. Additionally, we demonstrated that the majority of cataloged smells (97%) have at least mid-relevance levels, indicating their potential to hinder the readability, maintenance, or evolution of Elixir systems. Furthermore, we showed that most of these smells (54%) exhibit at least mid-prevalence levels, making them common in production code. These findings can assist developers, for instance, in establishing priorities for preventing and removing code smells.

- We also proposed a **comprehensive catalog of 82 refactorings for Elixir** (Chapter 4). To better organize this catalog, we categorized the refactorings into four distinct groups: 14 Elixir-Specific refactorings, 32 Functional refactorings, 11 Erlang-Specific refactorings, and 25 Traditional refactorings. Additionally, we revealed that most of the cataloged refactorings for Elixir (70.6%) are at least moderately prevalent, indicating their recurring use in production code. Furthermore, we showed that the vast majority of refactorings (92.7%) are at least moderately relevant, suggesting they have the potential to enhance the quality of Elixir systems. These findings indicate that developers should prioritize mastering the most prevalent refactorings first, as understanding these code transformations can save time during the code review process. Additionally, performing the most relevant refactorings first can help maximize improvements in code quality.

- We **correlated the cataloged code smells and refactorings for Elixir** (Chapter 5) and proposed **practical guidelines for removing each code smell in a disciplined manner using refactoring strategies in this language** (Appendix D). In total, we identified 176 relationships between code smells and corresponding refactorings for Elixir, which can be applied to eliminate these smells. More specifically, we found that all 35 code smells for Elixir have their removal assisted by at least one refactoring also cataloged for this language. Additionally, we showed that some refactorings can resolve multiple code smells, while 12 of the 82 cataloged refactorings are not associated with the removal of any known Elixir smell. Through these correlations, we also identified five new composite refactorings that are useful for removing code smells in Elixir. Furthermore, we demonstrated that traditional refactorings, originally proposed to enhance the quality of object-oriented systems [74], also play a significant role in eliminating code smells in Elixir.

- We created a **GitHub repository to document all the code smells in our catalog**. This documentation is formatted to be developer-friendly, containing, for instance, a description of the problem caused by each smell, along with code examples and textual descriptions to illustrate the occurrence of the code smell. This repository is available at `https://github.com/lucasvegi/Elixir-Code-Smells`. Es-

sentially, this artifact allows us to make public all the details of the sub-optimal structures reported in Chapter 3.

- We also created another **GitHub repository to document all the cataloged refactorings for Elixir**: `https://github.com/lucasvegi/Elixir-Refactorings`. For each refactoring, we provide a structured description of the main motivations for performing them and illustrate the resulting code from the refactoring, showing versions of it before and after the transformation. Therefore, developers can consult this repository to understand and apply the findings reported in Chapter 4 to their code.

- The investigations conducted in this thesis and their respective results have **fostered discussions about software quality among members of the Elixir developer community**. After sparking developers' interest and becoming one of the 60 most popular Elixir repositories on GitHub among over 100k existing ones, part of the content from our repository on code smells for Elixir was incorporated into the official Elixir documentation: `https://hexdocs.pm/elixir/what-anti-patterns.html`. Furthermore, since 2022, the studies conducted in this thesis have been discussed recurrently in major podcasts and conferences aimed at Elixir developers.

6.3 Future Work

Throughout the research conducted in this thesis, we identified some unexplored topics with the potential for significant future studies. These topics are outlined in the following paragraphs:

Metrics for detecting code smells in Elixir. As presented in Section 2.3, there are numerous studies and tools that rely on a combination of metrics such as Weighted Methods Per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), Coupling between Objects (CBO), Response For Class (RFC), and Lack of Cohesion of Methods (LCOM) to detect traditional code smells [20, 122, 123]. To the best of our knowledge, these studies and tools are focused on object-oriented code. Therefore, we claim that there is a lack of understanding regarding the effectiveness of using these classical metrics to detect Elixir-specific smells (Chapter 3). Thus, we suggest, as future work, the conduction of studies aimed at verifying whether these classical metrics can be used to detect code smells specific to a functional language like Elixir, and eventually propose new metrics tailored to this context.

Creation or adaptation of tools for detecting code smells in Elixir. Although in Section 2.3 we showed that there are more than 80 tools available in the literature focused on detecting code smells (*e.g.,* DECOR [129], PMD [70], JDeodorant [69], and JSpIRIT [202]), in Chapter 3 of this thesis, we also discussed the existence of few tools capable of detecting code smells in systems implemented in Elixir. More specifically, only three code smells from our catalog (Chapter 3) are automatically detected by Credo,[1] which is currently the most popular linter tool for Elixir. This motivates the development of new code smell detection tools for this language in future work. These tools for detecting code smells in Elixir can be based on a combination of strategies that utilize software metrics, as discussed in the previous topic, static code analysis like the one currently performed by Credo, or even the use of machine learning algorithms, as done by Aniche *et al.* [9] in a more generic context. Considering that Credo is open-source, instead of creating a new tool from scratch, a potential direction for future work would be to adapt Credo to detect a larger number of code smells for Elixir.

Creation and adaptation of tools for automated refactoring in Elixir. In Section 2.4, we presented many tools focused on automated refactoring in languages such as C# [90], SmallTalk [152], Java [61, 89, 134, 159], Swift [147], Scala [166], Erlang [112, 119, 156], Haskell [36, 78, 81, 161, 192], and OCaml [153]. However, in Chapter 4 of this thesis, we showed that developers working with Elixir still lack a more comprehensive tool for refactoring code in this language, as Styler,[2] the main tool for this purpose in Elixir, can perform only seven (out of the 82) refactorings from our catalog (Chapter 4). Thus, we suggest that future work could involve adapting Styler, an open-source tool, to incorporate more of the refactoring strategies cataloged in this thesis. Additionally, efforts could be directed toward developing new tools for refactoring Elixir code. These tools can be based on the analysis and transformation of the Elixir AST, potentially utilizing libraries such as Sourceror,[3] which facilitates AST manipulation. Another possibility involves creating refactoring tools for Elixir based on the *Language Server Protocol* (LSP).[4] Language Servers provide features like auto-completion, go-to-definition, find-all-references, and others for a specific language. Those features are normally implemented from scratch by each IDE for the language. LSP, created by Microsoft, standardizes how such servers and IDEs communicate, thus reducing redundant implementations of the same features across tools. Using LSP to build a refactoring tool for Elixir could be something especially promising, given that the core team

[1] http://credo-ci.org/

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

[3] https://github.com/doorgan/sourceror

[4] https://microsoft.github.io/language-server-protocol/

maintaining the language recently announced the development of the official Elixir Language Server, consolidating ongoing efforts to implement LSP for Elixir.[5] Lastly, another promising direction for future work could involve understanding how automated refactoring tools for Elixir can be enhanced through integration with LLMs, such as GPT-4.[6]

Catalog of composite refactorings for Elixir. Although we identified five composite refactorings for Elixir in the study described in Chapter 5 of this thesis, this was not the primary focus of the investigation. Considering that we were able to identify some of these coarse-grained source code transformations, even though it was only a secondary objective of our study, we believe that there are many more beyond these five composite refactorings that could be identified more efficiently in studies focused on understanding them. Thus, we suggest as future work creating a comprehensive catalog of composite refactorings for Elixir. One direction for conducting these studies could involve a partial adaptation of the methodology used by Brito *et al.* [32] to propose a catalog of composite refactorings focused on object-oriented code. In that study, the authors used the tool RefDiff [170] to mine the history of 10 well-known open-source projects on GitHub in search of composite refactorings performed on Java code. Since RefDiff currently does not have the capability to identify refactoring operations performed in the version history of systems developed in Elixir, and to the best of our knowledge, there is no tool with this ability, another future work associated with creating a catalog of composite refactorings for Elixir could involve adapting RefDiff, which is an open-source tool, by adding support for this language.

Catalog of test smells for Elixir. Test smells are bad programming practices that indicate potential problems in the design and implementation of automated software tests, potentially affecting the maintainability, coverage, and reliability of test code [172, 194]. Several studies in the literature have investigated the implications of test smells in specific contexts, such as Java [173], C# [145], and Python [91], among others. Although this thesis does not explore test-related smells in Elixir, we believe they may exist and warrant investigation in future work. As Elixir is a functional language and thus tends to have pure functions [15], we hypothesize that the manifestation of test smells in Elixir's test code may differ from those in object-oriented languages. For instance, there might be a lower incidence of non-deterministic tests, commonly referred to as flaky tests [80, 120].

Behavior preservation guarantees of refactorings for Elixir. Although in Chapter 5 of this thesis we empirically produced a practical guide on how to remove code

[5] https://elixir-lang.org/blog/2024/08/15/welcome-elixir-language-server-team/

[6] https://openai.com/index/gpt-4/

smells in Elixir using refactorings for this language, we did not aim to formally prove that all these code transformations fully preserve the original behavior of the programs. To achieve this, it would be necessary, for example, to apply a rigorous mathematical formalism to define syntactic and semantic equivalences in Elixir. Some studies focused on Erlang have used such approaches to prove that specific refactoring strategies preserve program behavior in that functional language [22, 23, 163]. Thus, we suggest that future work, analogous to those conducted for Erlang, should be carried out to ensure the behavior preservation of all Elixir code refactored according to the practical guidelines provided in this thesis (Appendix D).

Interrelation between code smells for Elixir. Although we proposed a comprehensive catalog of code smells for Elixir in this thesis (Chapter 3), we did not investigate the existence of interrelations between these sub-optimal code structures. However, some related works have already established interrelations between traditional code smells, such as those proposed by Fowler and Beck [74], intending to reduce the effort required to remove them [116, 175]. For instance, Liu *et al.* [116] demonstrated that removing instances of Duplicated code can also promote the elimination of Long function instances. Thus, it may be beneficial to prioritize the removal of Duplicated code over other smells that do not also result in the elimination of other types of smells. In light of this, we suggest future investigations to explore potential interrelations of this nature between the Elixir-specific smells from our catalog. We hypothesize that the understanding of these interrelations, together with the levels of relevance of the smells reported in Chapter 3, could serve as valuable indicators for developers when prioritizing which smell to remove first in their codebases. For example, by removing an Elixir-specific smell that not only has high relevance but also, when eliminated, leads to the disappearance of many other types of smells, developers could minimize the effort needed to achieve broader quality improvements in a system.

Quantify the impacts of refactorings for Elixir on software quality attributes. Although in this thesis we qualitatively indicated which software quality attributes (*e.g.,* reusability, readability, extensibility, etc.) are impacted by the refactorings in our catalog (Chapters 4 and 5), and we also quantified the perceptions of Elixir developers regarding the relevance of these code transformation strategies (Chapter 4), we did not analyze real-world Elixir projects to individually measure the magnitude of the impacts these Elixir-tailored refactorings have on the corresponding software quality attributes they affect. In related works, some authors have used metrics to quantify the impacts of traditional refactorings on software quality attributes. For instance, Almogahed *et al.* [7] quantified the impact of five traditional refactoring strategies on code reusability in a well-known open-source system.

Similarly, Weißgerber and Diehl [205] analyzed the history of open-source systems to quantify the effect of traditional refactorings on bug rates in the systems they studied. Therefore, we propose that future studies, akin to the aforementioned, be conducted specifically in the context of Elixir. These studies should focus on quantifying the impacts of the refactorings from our catalog (Chapter 4) on real-world Elixir projects, thus enabling a more precise and objective understanding of the extent of improvements or drawbacks resulting from the use of these transformations.

Use of eye tracking to evaluate developers' perception of the relevance of code smells and refactorings for Elixir. Although this thesis evaluates the relevance of code smells and refactorings for Elixir through questionnaires applied to developers (Chapters 3 and 4), studies conducted on code developed in Python [55] and C [54] suggest that eye tracking should be considered in investigations aimed at assessing the difficulty developers face when understanding the code they are working with. Specifically, by measuring the time spent by a developer to complete a task in the code, the number of attempts, and visual effort, eye tracking can quantify the level of difficulty developers encounter in maintaining a codebase [55]. Therefore, we hypothesize that eye tracking could be used in future work to assess the difficulty developers experience when maintaining Elixir code with code smells and their refactored versions, providing a complementary evaluation of developers' perceptions regarding the relevance of the smells and refactorings cataloged in this thesis.

Generalization of catalogs for the functional programming paradigm. As presented in Chapter 2, there are studies addressing the existence of code smells and refactorings specific to different functional languages. However, to the best of our knowledge, this thesis was the first study focused on proposing comprehensive catalogs of smells and refactorings for a functional language and correlating them. Considering that the investigations in this thesis were concentrated on the specific context of the functional language Elixir, it is not possible to affirm that all of our findings are applicable to other functional programming languages. In light of this, we conjecture that future work aiming to replicate the investigations of this thesis in the specific context of other functional languages like Clojure, F#, Scala, Julia, and others should be conducted. Following the completion of these new studies specific to other functional languages, an investigation aimed at identifying the intersections of their findings could finally lead to the generalization of code smells and refactorings characteristic of the functional paradigm, thus reaching results applicable to a broader scope, similar to those produced by Fowler and Beck [74] for the object-oriented programming paradigm.

References

[1] Abid, C., Alizadeh, V., Kessentini, M., Ferreira, T. N., and Dig, D. 30 years of software refactoring research: a systematic literature review. *ArXiv*, abs/2007.02194: 1–23, 2020. doi: https://doi.org/10.48550/arXiv.2007.02194.

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

[3] Agrahari, V., Shanbhag, S., Chimalakonda, S., and Rao, A. E. A catalogue of game-specific anti-patterns based on GitHub and game development stack exchange. *Journal of Systems and Software*, page 111789, 2023. doi: https://doi.org/10.1016/j.jss.2023.111789.

[4] Al Dallal, J. and Abdin, A. Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review. *IEEE Transactions on Software Engineering*, 44(1):44–69, 2018. doi: 10.1109/TSE.2017.2658573.

[5] Al-Fraihat, D., Sharrab, Y., Al-Ghuwairi, A., Sbaih, N., and Qahmash, A. Detecting refactoring type of software commit messages based on ensemble machine learning algorithms. *Scientific Reports*, 14(21367):1–20, 2024. doi: https://doi.org/10.1038/s41598-024-72307-0.

[6] Almeida, U. *Learn functional programming with Elixir: new foundations for a new world*. Pragmatic Bookshelf, 1 edition, 2018.

[7] Almogahed, A., Mahdin, H., Rejab, M. M., Alawadhi, A., Barraood, S. O., Othman, M., Al-Jamili, O., Almazroi, A. A., and Shaharudin, S. M. Code refactoring for software reusability: An experimental study. In *4th International Conference on Emerging Smart Technologies and Applications (eSmarTA)*, pages 1–6, 2024. doi: https://doi.org/10.1109/eSmarTA62850.2024.10638872.

[8] AlOmar, E. A., Venkatakrishnan, A., Mkaouer, M. W., Newman, C., and Ouni, A. How to refactor this code? an exploratory study on developer-ChatGPT refactoring conversations. In *21st International Conference on Mining Software Repositories (MSR)*, page 202–206, 2024. doi: https://doi.org/10.1145/3643991.3645081.

[9] Aniche, M., Maziero, E., Durelli, R., and Durelli, V. S. The effectiveness of supervised machine learning algorithms in predicting software refactoring. *IEEE Transactions on Software Engineering*, 48(04):1432–1450, 2022. doi: https://doi.ieeecomputersociety.org/10.1109/TSE.2020.3021736.

[10] Aranda, M., Oliveira, N., Soares, E., Ribeiro, M., Romão, D., Patriota, U., Gheyi, R., Souza, E., and Machado, I. A catalog of transformations to remove smells from natural language tests. In *28th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 7–16, 2024. doi: https://doi.org/10.1145/3661167.3661225.

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

[12] Arnaoudova, V., Di Penta, M., and Antoniol, G. Linguistic antipatterns: What they are and how developers perceive them. *Empirical Software Engineering*, 21(1): 104–158, 2016. doi: https://doi.org/10.1007/s10664-014-9350-8.

[13] Atwi, H., Lin, B., Tsantalis, N., Kashiwa, Y., Kamei, Y., Ubayashi, N., Bavota, G., and Lanza, M. PyRef: refactoring detection in Python projects. In *21st IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 136–141, 2021.

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

[15] Backfield, J. *Becoming functional: steps for transforming into a functional programmer*. O'Reilly Media, 1 edition, 2014.

[16] Backus, J. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, 1978. doi: https://doi.org/10.1145/359576.359579.

[17] Baltes, S. and Diehl, S. Worse than spam: Issues in sampling software developers. In *10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–6, 2016. doi: https://doi.org/10.1145/2961111.2962628.

[18] Barn, B., Barat, S., and Clark, T. Conducting systematic literature reviews and systematic mapping studies. In *10th Innovations in Software Engineering Conference (ISEC)*, page 212–213, 2017. doi: https://doi.org/10.1145/3021460.3021489.

[19] Barwell, A. D., Brown, C. M., and Hammond, K. Finding parallel functional pearls: Automatic parallel recursion scheme detection in Haskell functions via anti-unification. *Future Generation Computer Systems*, 79:669–686, 2018. doi: https://doi.org/10.1016/j.future.2017.07.024.

[20] Bavota, G., De Lucia, A., Di Penta, M., Oliveto, R., and Palomba, F. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software*, 107:1–14, 2015.

[21] Bellegarde, F. Notes for pipelines of transformations for ML. Technical report, Oregon Graduate Institute of Science & Technology, 1995.

[22] Bereczky, P., Horpácsi, D., and Thompson, S. A frame stack semantics for sequential Core Erlang. In *35th Symposium on Implementation and Application of Functional Languages (IFL)*, pages 1–13, 2024. doi: https://doi.org/10.1145/3652561.3652566.

[23] Bereczky, P., Horpácsi, D., and Thompson, S. A formalisation of Core Erlang, a concurrent actor language. *Acta Cybernetica*, 26(3):373–404, 2024. doi: https://doi.org/10.14232/actacyb.298977.

[24] Bibiano, A. C., Soares, V., Coutinho, D., Fernandes, E., Correia, J. a. L., Santos, K., Oliveira, A., Garcia, A., Gheyi, R., Fonseca, B., Ribeiro, M., Barbosa, C., and Oliveira, D. How does incomplete composite refactoring affect internal quality attributes? In *28th International Conference on Program Comprehension (ICPC)*, page 149–159, 2020. doi: https://doi.org/10.1145/3387904.3389264.

[25] Bibiano, A. C., Assunção, W. K. G., Coutinho, D., Santos, K., Soares, V., Gheyi, R., Garcia, A., Fonseca, B., Ribeiro, M., Oliveira, D., Barbosa, C., Marques, J. L., and Oliveira, A. Look ahead! revealing complete composite refactorings and their smelliness effects. In *37th IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 298–308, 2021. doi: https://doi.org/10.1109/ICSME52107.2021.00033.

[26] Bordignon, M. D. and Silva, R. A. Mutation operators for concurrent programs in Elixir. In *21st IEEE Latin-American Test Symposium (LATS)*, pages 1–6, 2020. doi: https://doi.org/10.1109/LATS49555.2020.9093675.

[27] Borrelli, A., Nardone, V., Di Lucca, G. A., Canfora, G., and Di Penta, M. Detecting video game-specific bad smells in Unity projects. In *17th International Conference on Mining Software Repositories (MSR)*, page 198–208, 2020. doi: https://doi.org/10.1145/3379597.3387454.

[28] Bosco, M., Cavoto, P., Ungolo, A., Muse, B. A., Khomh, F., Nardone, V., and Di Penta, M. UnityLint: A bad smell detector for Unity. In *31st IEEE/ACM*

International Conference on Program Comprehension (ICPC), pages 186–190, 2023. doi: https://doi.org/10.1109/ICPC58990.2023.00033.

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

[30] Bozó, I., Fördős, V., Horpácsi, D., Horváth, Z., Kozsik, T., Kőszegi, J., and Tóth, M. Refactorings to enable parallelization. In *15th Trends in Functional Programming (TFP)*, pages 104–121, 2015.

[31] Bozó, I. and Tóth, M. 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)*, pages 162–176, 2009.

[32] Brito, A., Hora, A., and Valente, M. T. Towards a catalog of composite refactorings. *Journal of Software: Evolution and Process*, 1:1–22, 2023. doi: https://doi.org/10.1002/smr.2530.

[33] Brito, R. and Valente, M. T. RefDiff4Go: Detecting refactorings in Go. In *14th Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS)*, pages 1–10, 2020.

[34] Brown, C. M. and Thompson, S. Clone detection and elimination for Haskell. In *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM)*, page 111–120, 2010. doi: https://doi.org/10.1145/1706356.1706378.

[35] Brown, C. M., Hammond, K., Danelutto, M., Kilpatrick, P., Schöner, H., and Breddin, T. Paraphrasing: Generating parallel programs using refactoring. In *10th Symposium on Formal Methods for Components and Objects (FMCO)*, pages 237–256, 2011. doi: https://doi.org/10.1007/978-3-642-35887-6_13.

[36] Brown, C. M., Li, H., and Thompson, S. An expression processor: A case study in refactoring Haskell programs. In Page, R., Horváth, Z., and Zsók, V., editors, *12th Trends in Functional Programming (TFP)*, volume 6546, pages 31–49, 2011.

[37] Brown, C. M., Loidl, H.-W., and Hammond, K. ParaForming: Forming parallel Haskell programs using novel refactoring techniques. In *12th International Conference on Trends in Functional Programming (TFP)*, pages 82–97, 2011. doi: https://doi.org/10.1007/978-3-642-32037-8_6.

[38] Brown, C. M., Danelutto, M., Hammond, K., Kilpatrick, P., and Elliott, A. Cost-directed refactoring for parallel Erlang programs. *International Journal of Parallel Programming*, 42:564–582, 2014.

[39] Brown, C. M. *Tool support for refactoring Haskell programs*. Phd thesis, University of Kent, UK, September 2008.

[40] Brown, W. J., Malveau, R. C., McCormick, H. W., and Mowbray, T. J. *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley and Sons, 1998.

[41] Budgen, D. and Brereton, P. Performing systematic literature reviews in software engineering. In *28th International Conference on Software Engineering (ICSE)*, pages 1051–1052, 2006. doi: https://doi.org/10.1145/1134285.1134500.

[42] Burstall, R. M. and Darlington, J. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, 1977. doi: https://doi.org/10.1145/321992.321996.

[43] Cedrim, D., Garcia, A., Mongiovi, M., Gheyi, R., Sousa, L., de Mello, R., Fonseca, B., Ribeiro, M., and Chávez, A. Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects. In *11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, page 465–475, 2017. doi: https://doi.org/10.1145/3106237.3106259.

[44] Chang, S. Laziness by need. In *Programming Languages and Systems (ESOP)*, pages 81–100, 2013.

[45] Chechina, N., MacKenzie, K., Thompson, S., Trinder, P., Boudeville, O., Fördős, V., Hoch, C., Ghaffari, A., and Hernandez, M. M. Evaluating scalable distributed Erlang for scalability and reliability. *IEEE Transactions on Parallel and Distributed Systems*, 28(8):2244–2257, 2017. doi: https://doi.org/10.1109/TPDS.2017.2654246.

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

[47] Chidamber, S. and Kemerer, C. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994. doi: https://doi.org/10.1109/32.295895.

[48] Church, A. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, 1932.

[49] Corbat, T., Felber, L., Stocker, M., and Sommerlad, P. Ruby refactoring plug-in for Eclipse. In *22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion (OOPSLA)*, pages 779–780, 2007. doi: https://doi.org/10.1145/1297846.1297884.

[50] Cowie, J. Detecting bad smells in Haskell. Technical report, University of Kent, UK, 2005.

[51] Cruz, D., Santana, A., and Figueiredo, E. Detecting bad smells with machine learning algorithms: An empirical study. In *3rd International Conference on Technical Debt (TechDebt)*, page 31–40, 2020. doi: https://doi.org/10.1145/3387906.3388618.

[52] Cruzes, D. S. and Dybå, T. Research synthesis in software engineering: A tertiary study. *Information and Software Technology*, 53(5):440–455, 2011. doi: https://doi.org/10.1016/j.infsof.2011.01.004.

[53] Cruzes, D. S. and Dybå, T. Recommended steps for thematic synthesis in software engineering. In *5th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 275–284, 2011. doi: https://doi.org/10.1109/ESEM.2011.36.

[54] da Costa, J. A. S., Gheyi, R., Ribeiro, M., Apel, S., Alves, V., Fonseca, B., Medeiros, F., and Garcia, A. Evaluating refactorings for disciplining #ifdef annotations: An eye tracking study with novices. *Empirical Software Engineering*, 26(92):1–35, 2021. doi: https://doi.org/10.1007/s10664-021-10002-8.

[55] da Costa, J. A. S., Gheyi, R., Castor, F., de Oliveira, P. R. F., Ribeiro, M., and Fonseca, B. Seeing confusion through a new lens: on the impact of atoms of confusion on novices' code comprehension. *Empirical Software Engineering*, 28(81):1–42, 2023. doi: https://doi.org/10.1007/s10664-023-10311-0.

[56] Dabic, O., Aghajani, E., and Bavota, G. Sampling projects in GitHub for MSR studies. In *18th IEEE/ACM International Conference on Mining Software Repositories (MSR)*, pages 560–564, 2021. doi: https://doi.org/10.1109/MSR52588.2021.00074.

[57] Dig, D. A refactoring approach to parallelism. *IEEE Software*, 28(1):17–22, 2011. doi: 10.1109/MS.2011.1.

[58] Dig, D., Comertoglu, C., Marinov, D., and Johnson, R. Automated detection of refactorings in evolving components. In *20th European Conference on Object-Oriented Programming (ECOOP)*, pages 404–428, 2006.

[59] Dong, Y., Li, Z., Tian, Y., Sun, C., Godfrey, M. W., and Nagappan, M. Bash in the wild: language usage, code smells, and bugs. *ACM Transactions on Software Engineering and Methodology*, 2022. doi: https://doi.org/10.1145/3517193.

[60] Drienyovszky, D., Horpácsi, D., and Thompson, S. Quickchecking refactoring tools. In *9th ACM SIGPLAN Workshop on Erlang*, page 75–80, 2010. doi: https://doi.org/10.1145/1863509.1863521.

[61] Eclipse, F. Eclipse IDE. Available at: `https://www.eclipse.org/`, 2023.

[62] Erwig, M. and Ren, D. An update calculus for expressing type-safe program updates. *Science of Computer Programming*, 67(2):199–222, 2007. doi: https://doi.org/10.1016/j.scico.2007.01.003.

[63] Fard, A. M. and Mesbah, A. Jsnose: detecting JavaScript code smells. In *13th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 116–125, 2013. doi: https://doi.org/10.1109/SCAM.2013.6648192.

[64] Farmer, A., Gill, A., Komp, E., and Sculthorpe, N. The HERMIT in the machine: A plugin for the interactive transformation of GHC core language programs. In *ACM SIGPLAN Symposium on Haskell (Haskell'12)*, pages 1–12, 2012. doi: https://doi.org/10.1145/2364506.2364508.

[65] Farmer, A., Sculthorpe, N., and Gill, A. Reasoning with the HERMIT: Tool support for equational reasoning on GHC core programs. In *ACM SIGPLAN Symposium on Haskell (Haskell'15)*, pages 23–34, 2015. doi: https://doi.org/10.1145/2804302.2804303.

[66] Fernandes, E., Oliveira, J., Vale, G., Paiva, T., and Figueiredo, E. A review-based comparative study of bad smell detection tools. In *20th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 1–12, 2016. doi: https://doi.org/10.1145/2915970.2915984.

[67] Ferreira, F. and Valente, M. T. Detecting code smells in React-based web apps. *Information and Software Technology*, 155:1–16, 2023. doi: https://doi.org/10.1016/j.infsof.2022.107111.

[68] Ferreira, F., Borges, H., and Valente, M. T. Refactoring React-based web apps. *Journal of Systems and Software*, 215:1–36, 2024. doi: https://doi.org/10.1016/j.jss.2024.112105.

[69] Fokaefs, M., Tsantalis, N., Stroulia, E., and Chatzigeorgiou, A. JDeodorant: identification and application of extract class refactorings. In *33rd International Conference on Software Engineering (ICSE)*, pages 1037–1039, 2011. doi: https://doi.org/10.1145/1985793.1985989.

[70] Fontana, F., Zanoni, M., Marino, A., and Mäntylä, M. Code smell detection: towards a machine learning-based approach. In *29th International Conference on*

Software Maintenance (ICSM), pages 396–399, 2013. doi: https://doi.org/10.1109/ICSM.2013.56.

[71] Fontana, F. A., Mäntylä, M. V., Zanoni, M., and Marino, A. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, 21(3):1143–1191, 2016. doi: https://doi.org/10.1007/s10664-015-9378-4.

[72] Fontana, F. A., Ferme, V., Marino, A., Walter, B., and Martenka, P. Investigating the impact of code smells on system's quality: an empirical study on systems of different application domains. In *29th IEEE International Conference on Software Maintenance (ICSM)*, pages 260–269, 2013. doi: https://doi.org/10.1109/ICSM.2013.37.

[73] Fördős, V. and Tóth, M. Identifying code clones with RefactorErl. *Acta Cybernetica*, 22(3):553–571, 2016. doi: https://doi.org/10.14232/actacyb.22.3.2016.1.

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

[75] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design patterns: Elements of reusable object-oriented software.* Pearson Education, 1 edition, 1994.

[76] Garousi, V., Felderer, M., and Mäntylä, M. V. Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. *Information and Software Technology*, 106(1):101–121, 2019. doi: https://doi.org/10.1016/j.infsof.2018.09.006.

[77] Gesi, J., Liu, S., Li, J., Ahmed, I., Nagappan, N., Lo, D., de Almeida, E. S., Kochhar, P. S., and Bao, L. Code smells in machine learning systems. *ArXiv*, arXiv:2203.00803:1–12, 2022. doi: https://doi.org/10.48550/arXiv.2203.00803.

[78] Gill, A. Introducing the Haskell equational reasoning assistant. In *ACM SIGPLAN Workshop on Haskell (Haskell'06)*, pages 108–109, 2006. doi: https://doi.org/10.1145/1159842.1159856.

[79] Golubev, Y., Kurbatova, Z., AlOmar, E. A., Bryksin, T., and Mkaouer, M. W. 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)*, pages 1303–1313, 2021. doi: https://doi.org/10.1145/3468264.3473924.

[80] Gruber, M., Lukasczyk, S., Kroiß, F., and Fraser, G. An empirical study of flaky tests in Python. In *14th IEEE Conference on Software Testing, Verification and*

Validation (ICST), pages 148–158, 2021. doi: https://doi.org/10.1109/ICST49551.2021.00026.

[81] Guttmann, W., Partsch, H., Schulte, W., and Vullinghs, T. Tool support for the interactive derivation of formally correct functional programs. *Journal of Universal Computer Science*, 9(2):173–188, 2003.

[82] Habchi, S., Hecht, G., Rouvoy, R., and Moha, N. Code smells in iOS apps: how do they compare to Android? In *4th IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 110–121, 2017. doi: https://doi.org/10.1109/MOBILESoft.2017.11.

[83] Hecht, G., Benomar, O., Rouvoy, R., Moha, N., and Duchien, L. Tracking the software quality of Android applications along their evolution. In *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 236–247, 2015. doi: https://doi.org/10.1109/ASE.2015.46.

[84] Héder, M., László, Z., and Sulyán, T. Another neat tool for refactoring Erlang programs. In *11th IASTED International Conference on Software Engineering and Applications (SEA)*, pages 336–341, 2007.

[85] Hudak, P. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, 21(3):359–411, 1989. doi: https://doi.org/10.1145/72551.72554.

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

[87] Ichtsis, A., Mittas, N., Ampatzoglou, A., and Chatzigeorgiou, A. Merging smell detectors: Evidence on the agreement of multiple tools. In *5th International Conference on Technical Debt (TechDebt)*, pages 61–65, 2022. doi: https://doi.org/10.1145/3524843.3528089.

[88] Janjic, V., Brown, C., Barwell, A., and Hammond, K. Refactoring for introducing and tuning parallelism for heterogeneous multicore machines in Erlang. *Concurrency and Computation: Practice and Experience*, 33(14):1–25, 2021.

[89] JetBrains. IntelliJ IDEA: the leading Java and Kotlin IDE. Available at: `https://www.jetbrains.com/idea/`, 2023.

[90] JetBrains. ReSharper: the visual studio extension for .NET developers. Available at: `https://www.jetbrains.com/resharper/`, 2023.

[91] Job, R. and Hora, A. How and why developers implement OS-specific tests. *Empirical Software Engineering*, 30(8):1–33, 2025. doi: https://doi.org/10.1007/s10664-024-10571-4.

[92] Johnson, B., Song, Y., Murphy-Hill, E., and Bowdidge, R. Why don't software developers use static analysis tools to find bugs? In *35th International Conference on Software Engineering (ICSE)*, pages 672–681, 2013. doi: https://doi.org/10.1109/ICSE.2013.6606613.

[93] Johnsson, T. Lambda lifting: Transforming programs to recursive equations. In *Functional Programming Languages and Computer Architecture (FPCA)*, pages 190–203, 1985.

[94] Jurić, S. *Elixir in action*. Manning, 3 edition, 2024.

[95] Kalhor, S., Keyvanpour, M. R., and Salajegheh, A. A systematic review of refactoring opportunities by software antipattern detection. *Automated Software Engineering*, 31(42):1–65, 2024. doi: https://doi.org/10.1007/s10515-024-00443-y.

[96] Kamei, F., Wiese, I., Lima, C., Polato, I., Nepomuceno, V., Ferreira, W., Ribeiro, M., Pena, C., Cartaxo, B., Pinto, G., and Soares, S. Grey literature in software engineering: a critical review. *Information and Software Technology*, 138(1):1–26, 2021. doi: https://doi.org/10.1016/j.infsof.2021.106609.

[97] Kim, M., Gee, M., Loh, A., and Rachatasumrit, N. Ref-Finder: a refactoring reconstruction tool based on logic query templates. In *8th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, page 371–372, 2010. doi: https://doi.org/10.1145/1882291.1882353.

[98] Király, R. Complexity metric based source code transformation of Erlang programs. In *Annales Mathematicae et Informaticae*, pages 29–44, 2013.

[99] Kitchenham, B. and Charters, S. Guidelines for performing systematic literature reviews in software engineering. Technical report, Keele University, July 2007.

[100] Kozsik, T., Csörnyei, Z., Horváth, Z., Király, R., Kitlei, R., Lövei, L., Nagy, T., Tóth, M., and Víg, A. Use cases for refactoring in Erlang. In *2nd Central European Functional Programming School (CEFP)*, pages 250–285, 2007. doi: https://doi.org/10.1007/978-3-540-88059-2_7.

[101] Kozsik, T., Tóth, M., Bozó, I., and Horváth, Z. Static analysis for divide-and-conquer pattern discovery. *Computing and Informatics*, 35(4):764–791, 2017.

[102] Kozsik, T., Tóth, M., and Bozó, I. Free the conqueror! refactoring divide-and-conquer functions. *Future Generation Computer Systems*, 79:687–699, 2018. URL https://doi.org/10.1016/j.future.2017.05.011.

[103] Kurbatova, Z., Kovalenko, V., Savu, I., Brockbernd, B., Andreescu, D., Anton, M., Venediktov, R., Tikhomirova, E., and Bryksin, T. RefactorInsight: Enhancing IDE representation of changes in Git with refactorings information. In *36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1276–1280, 2022. doi: https://doi.org/10.1109/ASE51524.2021.9678646.

[104] Lämmel, R. Towards generic refactoring. In *ACM SIGPLAN Workshop on Rule-Based Programming (RULE)*, page 15–28, 2002. doi: https://doi.org/10.1145/570186.570188.

[105] Lee, D. Y. A case study on refactoring in Haskell programs. In *33rd International Conference on Software Engineering (ICSE)*, pages 1164–1166, 2011. doi: https://doi.org/10.1145/1985793.1986030.

[106] Li, H. and Thompson, S. Comparative study of refactoring Haskell and Erlang programs. In *6th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 197–206, 2006. doi: https://doi.org/10.1109/SCAM.2006.8.

[107] Li, H. and Thompson, S. Clone detection and removal for Erlang/OTP within a refactoring environment. In *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM)*, pages 169–178, 2009. doi: https://doi.org/10.1145/1480945.1480971.

[108] Li, H. and Thompson, S. Refactoring support for modularity maintenance in Erlang. In *10th IEEE Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 157–166, 2010. doi: https://doi.org/10.1109/SCAM.2010.17.

[109] Li, H. and Thompson, S. A domain-specific language for scripting refactorings in Erlang. In de Lara, J. and Zisman, A., editors, *Fundamental Approaches to Software Engineering (FASE)*, volume 7212, pages 501–515, 2012.

[110] Li, H. and Thompson, S. Safe concurrency introduction through slicing. In *Workshop on Partial Evaluation and Program Manipulation (PEPM)*, page 103–113, 2015. doi: https://doi.org/10.1145/2678015.2682533.

[111] Li, H., Thompson, S., and Reinke, C. The Haskell refactorer, HaRe, and its API. *Electronic Notes in Theoretical Computer Science*, 141(4):29–34, 2005. doi: https://doi.org/10.1016/j.entcs.2005.02.053.

[112] Li, H., Thompson, S., Lövei, L., Horváth, Z., Kozsik, T., Víg, A., and Nagy, T. Refactoring Erlang programs. In *12th International Erlang/OTP User Conference (EUC)*, pages 1–10, 2006.

[113] Li, H., Thompson, S., Orosz, G., and Tóth, M. Refactoring with Wrangler, updated: Data and process refactorings, and integration with Eclipse. In *7th ACM SIGPLAN Workshop on ERLANG*, pages 61–72, 2008. doi: https://doi.org/10.1145/1411273.1411283.

[114] Li, H., Thompson, S., Lamela Seijas, P., and Francisco, M. A. Automating property-based testing of evolving web services. In *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM)*, page 169–180, 2014. doi: https://doi.org/10.1145/2543728.2543741.

[115] Li, W. and Shatnawi, R. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *Journal of Systems and Software*, 80(7):1120–1128, 2007. doi: https://doi.org/10.1016/j.jss.2006.10.018.

[116] Liu, H., Ma, Z., Shao, W., and Niu, Z. Schedule of bad smell detection and resolution: A new way to save effort. *IEEE Transactions on Software Engineering*, 38 (1):220–235, 2012. doi: https://doi.org/10.1109/TSE.2011.9.

[117] Lövei, L., Horváth, Z., Kozsik, T., and Király, R. Introducing records by refactoring. In *SIGPLAN Workshop on ERLANG Workshop*, pages 18–28, 2007. doi: https://doi.org/10.1145/1292520.1292524.

[118] Lövei, L., Horváth, Z., Kozsik, T., and Király, R. Introducing records by refactoring in Erlang programs. In *10th Symposium on Programming Languages and Software Tools (SPLST)*, pages 1–18, 2007.

[119] Lövei, L., Hoch, C., Köllö, H., Nagy, T., Nagyné Víg, A., Horpácsi, D., Kitlei, R., and Király, R. Refactoring module structure. In *7th ACM SIGPLAN Workshop on ERLANG*, pages 83–89, 2008. doi: https://doi.org/10.1145/1411273.1411285.

[120] Luo, Q., Hariri, F., Eloussi, L., and Marinov, D. An empirical analysis of flaky tests. In *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, page 643–653, 2014. doi: https://doi.org/10.1145/2635868.2635920.

[121] Mäntylä, M. V. and Lassenius, C. Subjective evaluation of software evolvability using code smells: an empirical study. *Empirical Software Engineering*, 11(3):395–431, 2006. doi: https://doi.org/10.1007/s10664-006-9002-8.

[122] Marinescu, R. Detecting design flaws via metrics in object-oriented systems. In *39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS)*, pages 173–182, 2001.

[123] Marinescu, R. Measurement and quality in object-oriented design. In *21st IEEE International Conference on Software Maintenance (ICSM)*, pages 701–704, 2005.

[124] Martin, R. C. *Functional design: Principles, patterns, and practices*. Addison-Wesley Professional, 1 edition, 2023.

[125] Mashiach, T., Sotto-Mayor, B., Kaminka, G., and Kalech, M. Clean++: Code smells extraction for C++. In *20th International Conference on Mining Software Repositories (MSR)*, pages 441–445, 2023. doi: https://doi.org/10.1109/MSR59073.2023.00066.

[126] McCarthy, J. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, 1960. doi: https://doi.org/10.1145/367177.367199.

[127] McKeeman, W. M. Differential testing for software. *Digital Technical Journal*, 10 (1):100–107, 1998.

[128] Meyer, B. *Object-oriented software construction*. Prentice Hall Englewood Cliffs, 2 edition, 1997.

[129] Moha, N., Gueheneuc, Y.-G., Duchien, L., and Le Meur, A.-F. DECOR: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, 2010. doi: https://doi.org/10.1109/TSE.2009.50.

[130] Mongiovi, M., Gheyi, R., Soares, G., Ribeiro, M., Borba, P., and Teixeira, L. Detecting overly strong preconditions in refactoring engines. *IEEE Transactions on Software Engineering*, 44(5):429–452, 2018. doi: https://doi.org/10.1109/TSE.2017.2693982.

[131] Murphy-Hill, E., Parnin, C., and Black, A. P. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, 2012. doi: https://doi.org/10.1109/TSE.2011.41.

[132] Nagappan, N., Ball, T., and Zeller, A. Mining metrics to predict component failures. In *28th International Conference on Software Engineering (ICSE)*, page 452–461, 2006. doi: https://doi.org/10.1145/1134285.1134349.

[133] Nardone, V., Muse, B. A., Abidi, M., Khomh, F., and Penta, M. D. Video game bad smells: What they are and how developers perceive them. *ACM Trans. Softw. Eng. Methodol.*, 32(4):1–35, 2023. doi: https://doi.org/10.1145/3563214.

[134] NetBeans.org. NetBeans IDE. Available at: `http://www.netbeans.org/`, 2023.

[135] Nunes, H. G., Vegi, L. F. M., Cruz, V. P. G., and Figueiredo, E. Democracia em xeque: um estudo comparativo sobre detecção de code smells. In *10th Workshop de Visualização, Evolução e Manutenção de Software (VEM)*, pages 11–15, 2022. doi: https://doi.org/10.5753/vem.2022.226562.

[136] Oizumi, W., Garcia, A., da Silva Sousa, L., Cafeo, B., and Zhao, Y. Code anomalies flock together: exploring code anomaly agglomerations for locating design problems. In *38th International Conference on Software Engineering (ICSE)*, page 440–451, 2016. doi: https://doi.org/10.1145/2884781.2884868.

[137] Oizumi, W. N., Garcia, A. F., Colanzi, T. E., Ferreira, M., and von Staa, A. When code-anomaly agglomerations represent architectural problems? an exploratory study. In *28th Brazilian Symposium on Software Engineering (SBSE)*, pages 91–100, 2014. doi: https://doi.org/10.1109/SBES.2014.18.

[138] Olbrich, S. M., Cruzes, D. S., and Sjøberg, D. I. Are all code smells harmful? a study of God Classes and Brain Classes in the evolution of three open source systems. In *26th IEEE International Conference on Software Maintenance (ICSM)*, pages 1–10, 2010. doi: https://doi.org/10.1109/ICSM.2010.5609564.

[139] Oliveira, J., Gheyi, R., Mongiovi, M., Soares, G., Ribeiro, M., and Garcia, A. Revisiting the refactoring mechanics. *Information and Software Technology*, 110: 136–138, 2019. doi: https://doi.org/10.1016/j.infsof.2019.03.002.

[140] Oliveira, J., Gheyi, R., Teixeira, L., Ribeiro, M., Leandro, O., and Fonseca, B. Towards a better understanding of the mechanics of refactoring detection tools. *Information and Software Technology*, 162:107273, 2023. doi: https://doi.org/10.1016/j.infsof.2023.107273.

[141] Opdyke, W. F. *Refactoring object-oriented frameworks*. Phd thesis, University of Illinois at Urbana-Champaign, USA, August 1992.

[142] Page, L., Brin, S., Motwani, R., and Winograd, T. The PageRank citation ranking: bringing order to the Web. Technical report, Stanford InfoLab, November 1999.

[143] Palomba, F., Bavota, G., Penta, M. D., Oliveto, R., and Lucia, A. D. Do they really smell bad? a study on developers' perception of bad code smells. In *30th*

IEEE International Conference on Software Maintenance and Evolution (ICSME), page 101–110, 2014. doi: https://doi.org/10.1109/ICSME.2014.32.

[144] Partsch, H. and Steinbrüggen, R. Program transformation systems. *ACM Comput. Surv.*, 15(3):199–236, 1983. doi: https://doi.org/10.1145/356914.356917.

[145] Paul, P. P., Akanda, M. T., Ullah, M. R., Mondal, D., Chowdhury, N. S., and Tawsif, F. M. xNose: A test smell detector for C#. In *IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE)*, page 370–371, 2024. doi: https://doi.org/10.1145/3639478.3643116.

[146] Punt, L., Visscher, S., and Zaytsev, V. The A?B*A pattern: undoing style in CSS and refactoring opportunities it presents. In *32nd IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 67–77, 2016. doi: https://doi.org/10.1109/ICSME.2016.73.

[147] Ramanathan, M. K., Clapp, L., Barik, R., and Sridharan, M. Piranha: Reducing feature flag debt at Uber. In *IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 221–230, 2020.

[148] Rasool, G. and Arshad, Z. A review of code smell mining techniques. *Journal of Software: Evolution and Process*, 27(11):867–895, 2015. doi: https://doi.org/10.1002/smr.1737.

[149] Ratzinger, J., Sigmund, T., and Gall, H. C. On the relation of refactorings and software defect prediction. In *5th International Working Conference on Mining Software Repositories (MSR)*, page 35–38, 2008. doi: https://doi.org/10.1145/1370750.1370759.

[150] Reimann, J., Brylski, M., and Aßmann, U. A tool-supported quality smell catalogue for Android developers. In *Modellbasierte und modellgetriebene Softwaremodernisierung (MMSM)*, page 1–2, 2014.

[151] Riel, A. J. *Object-oriented design heuristics*. Addison-Wesley, 1 edition, 1996.

[152] Roberts, D., Brant, J., and Johnson, R. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems*, 3(4):253–263, 1997.

[153] Rowe, R. N. S., Férée, H., Thompson, S., and Owens, S. ROTOR: A tool for renaming values in OCaml's module system. In *3rd International Workshop on Refactoring (IWOR)*, pages 27–30, 2019. doi: https://doi.org/10.1109/IWoR.2019.00013.

[154] Rowe, R. N. S., Férée, H., Thompson, S., and Owens, S. Characterising renaming within OCaml's module system: Theory and implementation. In *40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 950–965, 2019. doi: https://doi.org/10.1145/3314221.3314600.

[155] Saboury, A., Musavi, P., Khomh, F., and Antoniol, G. An empirical study of code smells in JavaScript projects. In *24th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 294–305, 2017. doi: https://doi.org/10.1109/SANER.2017.7884630.

[156] Sagonas, K. and Avgerinos, T. Automatic refactoring of Erlang programs. In *11th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 13–24, 2009. doi: https://doi.org/10.1145/1599410.1599414.

[157] Sagonas, K. and Luna, D. Gradual typing of Erlang programs: A Wrangler experience. In *7th ACM SIGPLAN Workshop on ERLANG*, pages 73–82, 2008. doi: https://doi.org/10.1145/1411273.1411284.

[158] Santos, G., Santana, A., Vale, G., and Figueiredo, E. Yet another model! a study on model's similarities for defect and code smells. In *26th International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 282–305, 2023. doi: https://doi.org/10.1007/978-3-031-30826-0_16.

[159] Schaefer, M. and de Moor, O. Specifying and implementing refactorings. In *ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, page 286–301, 2010. doi: https://doi.org/10.1145/1869459.1869485.

[160] Scott, M. L. *Programming language pragmatics*. Morgan Kaufmann, 4 edition, 2015.

[161] Sculthorpe, N., Farmer, A., and Gill, A. The HERMIT in the tree - mechanizing program transformations in the GHC core language. In Hinze, R., editor, *25th Symposium on Implementation and Application of Functional Languages (IFL). Lecture Notes in Computer Science*, volume 8241, pages 86–103, 2013.

[162] Seijas, P. L. and Thompson, S. Identifying and introducing interfaces and callbacks using Wrangler. In *28th Symposium on the Implementation and Application of Functional Programming Languages (IFL)*, pages 1–13, 2016. doi: https://doi.org/10.1145/3064899.3064909.

[163] Seres, B., Horpácsi, D., and Thompson, S. Is this really a refactoring? automated equivalence checking for Erlang projects. In *23rd ACM SIGPLAN International Workshop on Erlang*, page 55–66, 2024. doi: https://doi.org/10.1145/3677995.3678194.

[164] Sharma, T., Suryanarayana, G., and Samarthyam, G. Challenges to and solutions for refactoring adoption: An industrial perspective. *IEEE Software*, 32(6):44–51, 2015. doi: https://doi.org/10.1109/MS.2015.105.

[165] Sharma, T., Fragkoulis, M., and Spinellis, D. Does your configuration code smell? In *13th IEEE/ACM Working Conference on Mining Software Repositories (MSR)*, pages 189–200, 2016.

[166] Sherwany, A., Zaza, N., and Nystrom, N. A refactoring library for Scala compiler extensions. In Franke, B., editor, *24th International Conference on Compiler Construction (CC). Lecture Notes in Computer Science*, volume 9031, pages 31–48, 2015.

[167] Siegel, S. and Castellan, J. *Nonparametric statistics for the behavioral sciences*. McGraw-Hill International Editions, 2 edition, 1988.

[168] Silva, D. and Valente, M. T. RefDiff: Detecting refactorings in version histories. In *14th International Conference on Mining Software Repositories (MSR)*, pages 1–11, 2017.

[169] Silva, D., Tsantalis, N., and Valente, M. T. Why we refactor? confessions of GitHub contributors. In *24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 858–870, 2016. doi: https://doi.org/10.1145/2950290.2950305.

[170] Silva, D., da Silva, J. P., Santos, G., Terra, R., and Valente, M. T. RefDiff 2.0: A multi-language refactoring detection tool. *IEEE Transactions on Software Engineering*, 47(12):2786–2802, 2021. doi: https://doi.org/10.1109/TSE.2020.2968072.

[171] Singh, S. and Kaur, S. A systematic literature review: Refactoring for disclosing code smells in object oriented software. *Ain Shams Engineering Journal*, 9(4):2129–2151, 2018. doi: https://doi.org/10.1016/j.asej.2017.03.002.

[172] Soares, E., Aranda, M., Oliveira, N., Ribeiro, M., Gheyi, R., Souza, E., Machado, I., Santos, A., Fonseca, B., and Bonifacio, R. Manual tests do smell! Cataloging and identifying natural language test smells. In *17th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11, 2023. doi: https://doi.org/10.1109/ESEM56168.2023.10304800.

[173] Soares, E., Ribeiro, M., Gheyi, R., Amaral, G., and Santos, A. Refactoring test smells with JUnit 5: Why should developers keep up-to-date? *IEEE Transactions on Software Engineering*, 49(3):1152–1170, 2023. doi: https://doi.org/10.1109/TSE.2022.3172654.

[174] Soares, G., Mongiovi, M., and Gheyi, R. Identifying overly strong conditions in refactoring implementations. In *27th IEEE International Conference on Software Maintenance (ICSM)*, pages 173–182, 2011.

[175] Sobrinho, E. and Maia, M. On the interplay of smells large class, complex class and duplicate code. In *35th Brazilian Symposium on Software Engineering (SBSE)*, page 64–73, 2021. doi: https://doi.org/10.1145/3474624.3474716.

[176] Sobrinho, E., De Lucia, A., and Maia, M. A systematic literature review on bad smells–5 w's: which, when, what, who, where. *IEEE Transactions on Software Engineering*, 47(1):17–66, 2021. doi: https://doi.org/10.1109/TSE.2018.2880977.

[177] Soh, Z., Yamashita, A., Khomh, F., and Guéhéneuc, Y.-G. Do code smells impact the effort of different maintenance programming activities? In *23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 393–402, 2016. doi: https://doi.org/10.1109/SANER.2016.103.

[178] Sousa, L., Cedrim, D., Garcia, A., Oizumi, W., Bibiano, A. C., Oliveira, D., Kim, M., and Oliveira, A. Characterizing and identifying composite refactorings: Concepts, heuristics and patterns. In *17th International Conference on Mining Software Repositories (MSR)*, pages 186–197, 2020. doi: https://doi.org/10.1145/3379597.3387477.

[179] Swaine, M. *Functional programming: a PragPub anthology: exploring Clojure, Elixir, Haskell, Scala, and Swift.* Pragmatic Bookshelf, 1 edition, 2017.

[180] Taibi, D. and Lenarduzzi, V. On the definition of microservice bad smells. *IEEE Software*, 35(3):56–62, 2018. doi: https://doi.org/10.1109/MS.2018.2141031.

[181] Taibi, D., Janes, A., and Lenarduzzi, V. How developers perceive smells in source code: a replicated study. *Information and Software Technology*, 92(1):223–235, 2017. doi: https://doi.org/10.1016/j.infsof.2017.08.008.

[182] Taylor, R. and Derrick, J. mu2: A refactoring-based mutation testing framework for Erlang. In *27th International Conference on Testing Software and Systems (ICTSS)*, pages 178–193, 2015.

[183] Thomas, D. *Programming Elixir |> 1.6: functional |> concurrent |> pragmatic |> fun.* Pragmatic Bookshelf, 1 edition, 2018.

[184] Thompson, S. Refactoring functional programs. In *5th International Summer School on Advanced Functional Programming (AFP)*, pages 331–357, 2005.

[185] Thompson, S. *Haskell: the craft of functional programming.* Addison-Wesley Professional, 3 edition, 2023.

[186] Thompson, S. and Reinke, C. A case study in refactoring functional programs. In *7th Brazilian Symposium on Programming Languages (SBLP)*, pages 1–16, 2003.

[187] Thompson, S., Horpacsi, D., and Koszegi, J. Towards trustworthy refactoring in Erlang. In *4th International Workshop on Verification and Program Transformation (VPT)*, pages 83–103, 2016.

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

[189] Tóth, M., Bozó, I., and Kozsik, T. Pattern candidate discovery and parallelization techniques. In *29th Symposium on the Implementation and Application of Functional Programming Languages (IFL)*, pages 1–26, 2017. doi: https://doi.org/10.1145/3205368.3205369.

[190] Tsantalis, N., Mansouri, M., Eshkevari, L. M., Mazinanian, D., and Dig, D. Accurate and efficient refactoring detection in commit history. In *40th international conference on software engineering (ICSE)*, pages 483–494, 2018. doi: https://doi.org/10.1145/3180155.3180206.

[191] Tsantalis, N., Ketkar, A., and Dig, D. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering*, 48(3):930–950, 2022. doi: https://doi.org/10.1109/TSE.2020.3007722.

[192] Tullsen, M. A. *PATH, a program transformation system for Haskell*. Phd thesis, Yale University, USA, May 2002.

[193] Valente, M. T. *Engenharia de software moderna - princípios e práticas para desenvolvimento de software com produtividade*. Editora: Independente, 1 edition, 2020.

[194] van Deursen, A., Moonen, L., van den Bergh, A., and Kok, G. Refactoring test code. In *2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001)*, pages 92–95, 2001.

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

[196] Vegi, L. F. M. and Valente, M. T. Code smells in Elixir: early results from a grey literature review. In *30th International Conference on Program Comprehension (ICPC) - ERA track*, pages 580–584, 2022. doi: https://doi.org/10.1145/3524610.3527881.

[197] Vegi, L. F. M. and Valente, M. T. Catalog of Elixir refactorings. Available at: `https://github.com/lucasvegi/Elixir-Refactorings`, 2023.

[198] Vegi, L. F. M. and Valente, M. T. Understanding code smells in Elixir functional language - Replication Package. Available at: `https://doi.org/10.5281/zenodo.7430258`, 2023.

[199] Vegi, L. F. M. and Valente, M. T. Understanding code smells in Elixir functional language. *Empirical Software Engineering*, 28(102):1–32, 2023. doi: https://doi.org/10.1007/s10664-023-10343-6.

[200] Vegi, L. F. M. and Valente, M. T. Towards a catalog of refactorings for Elixir. In *39th International Conference on Software Maintenance and Evolution (ICSME) - NIER track*, pages 358–362, 2023. doi: https://doi.org/10.1109/ICSME58846.2023.00045.

[201] Vegi, L. F. M. and Valente, M. T. Understanding refactorings in Elixir functional language - Replication Package. Available at: `https://doi.org/10.5281/zenodo.11372758`, 2024.

[202] Vidal, S., Vazquez, H., Diaz-Pace, J., Marcos, C., Garcia, A., and Oizumi, W. JSpIRIT: a flexible tool for the analysis of code smells. In *34th International Conference of the Chilean Computer Science Society (SCCC)*, pages 1–6, 2015. doi: https://doi.org/10.1109/SCCC.2015.7416572.

[203] Villavicencio, G. A bottom-up approach to understand functional programs. In *4th International C* Conference on Computer Science and Software Engineering (C3S2E)*, page 111–120, 2011. doi: https://doi.org/10.1145/1992896.1992910.

[204] Walpole, R. E., Myers, R. H., Myers, S. L., and Ye, K. *Probability & statistics for engineers and scientists*. Pearson Education, 8 edition, 2007.

[205] Weißgerber, P. and Diehl, S. Are refactorings less error-prone than other changes? In *3rd International Workshop on Mining Software Repositories (MSR)*, page 112–118, 2006. doi: https://doi.org/10.1145/1137983.1138011.

[206] White, M., Tufano, M., Vendome, C., and Poshyvanyk, D. Deep learning code fragments for code clone detection. In *31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 87–98, 2016.

[207] Wohlin, C. 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)*, pages 1–10, 2014. doi: https://doi.org/10.1145/2601248.2601268.

[208] Wohlin, C., Runeson, P., Hst, M., Ohlsson, M. C., Regnell, B., and Wessln, A. *Experimentation in Software Engineering*. Springer, 2012.

[209] Xing, Z. and Stroulia, E. Refactoring practice: How it is and how it should be supported - an Eclipse case study. In *22nd IEEE International Conference on Software Maintenance (ICSM)*, pages 458–468, 2006.

[210] Yamashita, A. and Moonen, L. Do code smells reflect important maintainability aspects? In *28th IEEE International Conference on Software Maintenance (ICSM)*, pages 306–315, 2012. doi: https://doi.org/10.1109/ICSM.2012.6405287.

[211] Yamashita, A. and Moonen, L. Do developers care about code smells? an exploratory survey. In *20th Working Conference on Reverse Engineering (WCRE)*, pages 242–251, 2013. doi: https://doi.org/10.1109/WCRE.2013.6671299.

[212] Yamashita, A. and Moonen, L. To what extent can maintenance problems be predicted by code smell detection? an empirical study. *Information and Software Technology*, 55(12):2223–2242, 2013. doi: https://doi.org/10.1016/j.infsof.2013.08.002.

[213] Zhang, H., Cruz, L., and van Deursen, A. Code smells for machine learning applications. In *1st International Conference on AI Engineering: Software Engineering for AI (CAIN)*, page 217–228, 2022. doi: https://doi.org/10.1145/3522664.3528620.

[214] Zhang, H., Zhou, X., Huang, X., Huang, H., and Babar, M. A. An evidence-based inquiry into the use of grey literature in software engineering. In *42nd ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 1422–1434, 2020. doi: https://doi.org/10.1145/3377811.3380336.

[215] Zhang, Z., Xing, Z., Xia, X., Xu, X., and Zhu, L. Making Python code idiomatic by automatic refactoring non-idiomatic Python code with pythonic idioms. In *30th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, page 696–708, 2022. doi: https://doi.org/10.1145/3540250.3549143.

[216] Zhang, Z., Xing, Z., Zhao, D., Xu, X., Zhu, L., and Lu, Q. Automated refactoring of non-idiomatic Python code with pythonic idioms. *IEEE Transactions on Software Engineering*, pages 1–22, 2024. doi: https://doi.org/10.1109/TSE.2024.3420886.

[217] Zhao, J. On refactoring quantum programs in Q#. In *4th IEEE International Conference on Quantum Computing and Engineering (QCE)*, pages 169–172, 2023. doi: https://doi.org/10.1109/QCE57702.2023.10203.

Appendix A

Documents of the Survey on Code Smells in Elixir

In this appendix, we present the instruments used to perform the validation of our catalog of code smells for Elixir (Chapter 3). In Section A.1, we introduced the Free and Enlightened Consent document that should be signed by all participants of our survey before responding to our questionnaire on code smells in Elixir. In Section A.2, we presented each of the four versions of our questionnaire.

A.1 Free and Enlightened Consent

Title: Code Smells specific to systems implemented in the Elixir functional language

Institution: DCC / ICEx / UFMG

Responsible researchers:

Lucas Vegi (lucasvegi@dcc.ufmg.br)
Doctoral student at the Department of Computer Science at UFMG

Prof. Marco Túlio Valente (mtov@dcc.ufmg.br)
Associate professor at the Department of Computer Science at UFMG

Introduction: This "Free and Enlightened Consent" contains information about our research. If you have any questions, do not hesitate to ask the responsible researchers.

Evaluation goals: This study aims to understand the main code smells that can occur in Elixir systems.

Survey information: We will ask you questions about your demographics, positions held, experience with Elixir, and your perception of code smells in Elixir.

Data collection and use: The data will be collected through this Google Form. It

is estimated that each participant will need a maximum of 25 minutes to complete the answers. This data will be used to promote good practices to improve the quality of code implemented in Elixir. The identities of all participants, as well as their responses, will be kept confidential. You may choose to receive the preliminary results of the study, with the anonymity of data and participants preserved. The results will be published and presented at conferences and scientific journals without revealing the identity of the participants.

If you decide not to participate in the research: You are free to refuse to participate or withdraw your consent at any time. Your decision will not affect any relationship with the evaluators, professors, or the institution responsible for this research.

Compensation: Your participation is voluntary and unpaid. In addition, you will not be charged for participating in this research.

If you have any problems or any other questions about the research: You can contact Lucas Vegi at any time at lucasvegi@dcc.ufmg.br.

If you have questions about the ethical aspects of this research: Contact the Research Ethics Committee of the Federal University of Minas Gerais (COEP-UFMG). Address: Av. Antônio Carlos, 6627. Administrative Unit II – 2nd floor - Room 2005. Pampulha Campus. Belo Horizonte – MG, Brazil. CEP: 31270-901. E-mail: coep@prpq.ufmg.br. Phone: +55 (31) 3409-4592. Opening hours: 09:00 AM to 11:00 AM; 02:00 PM to 04:00 PM (UTC-3).

Risks and measures to minimize them: When answering the questions, you may feel uncomfortable with questions that can bring back bad memories, fear of not knowing the answer, or even being identified. If you feel uncomfortable, you can pause completing the questionnaire and withdraw from participation at any time. The guarantee of data confidentiality ensured by the researchers is limited to the privacy policies of the virtual environment used for data collection (Google Forms). According to Brazilian legislation (Art. 9 of Resolution nº 510/16 of the National Health Council), in case of any damages resulting from your participation in this research, you will have the right to be compensated, under the terms of the Law.

Benefits: This research aims to promote good practices for improving the quality of systems implemented in Elixir, thus justifying its risks.

This survey will be carried out entirely remotely. In this way, **this consent term will be made available in digital copy through this Google Form**. A copy of this digital consent will be kept by the responsible researchers and you will receive a digital copy via email after signing the term.

Participant e-mail: _____

Participant full name: _____

Free and Enlightened Consent (voluntary agreement):

☑ I declare that I have read the document mentioned above and that I agree to participate as a volunteer, authorizing the anonymous use of the data generated by my participation for the academic purposes described above.

A.2 Survey Questions

Table A.1: Demographic questions (equals across all questionnaire versions)

Topic	Questions
Demographics	**Select your country:** [Dropdown list...]
	Your city: _____
	Elixir experience time: [Less than 1 year | Between 1 and 3 years | More than 3 years]
	How many different projects have you participated in using Elixir? [Only one project | Between 2 and 4 projects | More than 4 projects]

Table A.2: Questionnaire A - Perceptions on traditional code smells in Elixir

<table>
<tr><th>Topic</th><th>Questions</th></tr>
<tr><td>Perceptions on code smells in Elixir</td><td>(Short overview of the concept of traditional code smells)</td></tr>
<tr><td></td><td>∗ How often does such smells occur in the Elixir systems you have worked with?
(1 = it is very rare; 5 = it is very common)
<table><tr><th>Smell (in random order)</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th></tr><tr><td>Inappropriate Intimacy
[link to description and code example]</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td></tr><tr><td>Large Class
[link to description and code example]</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td></tr><tr><td>Speculative Generality
[link to description and code example]</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td></tr></table></td></tr>
<tr><td></td><td>∗ How relevant are these smells in Elixir systems (evaluated as their potential to have a negative impact on maintainability, comprehensibility, and evolution)?
(1 = very low impact and relevance; 5 = high impact and relevance)
<table><tr><th>Smell (in random order)</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th></tr><tr><td>Inappropriate Intimacy
[link to description and code example]</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td></tr><tr><td>Large Class
[link to description and code example]</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td></tr><tr><td>Speculative Generality
[link to description and code example]</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td></tr></table></td></tr>
</table>

∗: *Optional answers (one per line).*

Table A.3: Questionnaire B - Perceptions on traditional code smells in Elixir

<table>
<tr><th>Topic</th><th>Questions</th></tr>
<tr><td>Perceptions on code smells in Elixir</td><td>(Short overview of the concept of traditional code smells)</td></tr>
<tr><td></td><td>∗ How often does such smells occur in the Elixir systems you have worked with?
(1 = it is very rare; 5 = it is very common)
<table>
<tr><th>Smell (in random order)</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th></tr>
<tr><td>Duplicated Code
[link to description and code example]</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td></tr>
<tr><td>Feature Envy
[link to description and code example]</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td></tr>
<tr><td>Switch Statements
[link to description and code example]</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td></tr>
</table></td></tr>
<tr><td></td><td>∗ How relevant are these smells in Elixir systems (evaluated as their potential to have a negative impact on maintainability, comprehensibility, and evolution)?
(1 = very low impact and relevance; 5 = high impact and relevance)
<table>
<tr><th>Smell (in random order)</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th></tr>
<tr><td>Duplicated Code
[link to description and code example]</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td></tr>
<tr><td>Feature Envy
[link to description and code example]</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td></tr>
<tr><td>Switch Statements
[link to description and code example]</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td></tr>
</table></td></tr>
</table>

∗: *Optional answers (one per line).*

Table A.4: Questionnaire C - Perceptions on traditional code smells in Elixir

<table>
<tr><th>Topic</th><th>Questions</th></tr>
<tr><td>Perceptions on code smells in Elixir</td><td>(Short overview of the concept of traditional code smells)</td></tr>
<tr><td></td><td>∗ How often does such smells occur in the Elixir systems you have worked with?
(1 = it is very rare; 5 = it is very common)
<table>
<tr><th>Smell (in random order)</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th></tr>
<tr><td>Comments
[link to description and code example]</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td></tr>
<tr><td>Long Parameter List
[link to description and code example]</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td></tr>
<tr><td>Primitive Obsession
[link to description and code example]</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td></tr>
</table></td></tr>
<tr><td></td><td>∗ How relevant are these smells in Elixir systems (evaluated as their potential to have a negative impact on maintainability, comprehensibility, and evolution)?
(1 = very low impact and relevance; 5 = high impact and relevance)
<table>
<tr><th>Smell (in random order)</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th></tr>
<tr><td>Comments
[link to description and code example]</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td></tr>
<tr><td>Long Parameter List
[link to description and code example]</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td></tr>
<tr><td>Primitive Obsession
[link to description and code example]</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td></tr>
</table></td></tr>
</table>

∗: *Optional answers (one per line).*

Table A.5: Questionnaire D - Perceptions on traditional code smells in Elixir

<table>
<tr><th>Topic</th><th>Questions</th></tr>
<tr><td>Perceptions on code smells in Elixir</td><td>(Short overview of the concept of traditional code smells)</td></tr>
<tr><td></td><td>∗ How often does such smells occur in the Elixir systems you have worked with?
(1 = it is very rare; 5 = it is very common)
<table>
<tr><th>Smell (in random order)</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th></tr>
<tr><td>Long Function
[link to description and code example]</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td></tr>
<tr><td>Shotgun Surgery
[link to description and code example]</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td></tr>
<tr><td>Divergent Change
[link to description and code example]</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td></tr>
</table></td></tr>
<tr><td></td><td>∗ How relevant are these smells in Elixir systems (evaluated as their potential to have a negative impact on maintainability, comprehensibility, and evolution)?
(1 = very low impact and relevance; 5 = high impact and relevance)
<table>
<tr><th>Smell (in random order)</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th></tr>
<tr><td>Long Function
[link to description and code example]</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td></tr>
<tr><td>Shotgun Surgery
[link to description and code example]</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td></tr>
<tr><td>Divergent Change
[link to description and code example]</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td></tr>
</table></td></tr>
</table>

∗: *Optional answers (one per line).*

Table A.6: Questionnaire A - Perceptions on Elixir-Specific code smells

Topic	Questions
Perceptions on code smells in Elixir	*(Short overview of the concept of Elixir-Specific code smells)*
	∗ **How often does such smells occur in the Elixir systems you have worked with?**

(1 = it is very rare; 5 = it is very common)

Smell (in random order)	1	2	3	4	5
GenServer Envy [link to description and code example]	◎	◎	◎	◎	◎
Untested polymorphic behaviors [link to description and code example]	◎	◎	◎	◎	◎
Data manipulation by migration [link to description and code example]	◎	◎	◎	◎	◎
Using App Configuration for libraries [link to description and code example]	◎	◎	◎	◎	◎
Working with invalid data [link to description and code example]	◎	◎	◎	◎	◎
Modules with identical names [link to description and code example]	◎	◎	◎	◎	◎

∗ **How relevant are these smells in Elixir systems (evaluated as their potential to have a negative impact on maintainability, comprehensibility, and evolution)?**

(1 = very low impact and relevance; 5 = high impact and relevance)

Smell (in random order)	1	2	3	4	5
GenServer Envy [link to description and code example]	◎	◎	◎	◎	◎
Untested polymorphic behaviors [link to description and code example]	◎	◎	◎	◎	◎
Data manipulation by migration [link to description and code example]	◎	◎	◎	◎	◎
Using App Configuration for libraries [link to description and code example]	◎	◎	◎	◎	◎
Working with invalid data [link to description and code example]	◎	◎	◎	◎	◎
Modules with identical names [link to description and code example]	◎	◎	◎	◎	◎

∗: *Optional answers (one per line).*

Table A.7: Questionnaire B - Perceptions on Elixir-Specific code smells

<table>
<tr><th>Topic</th><th>Questions</th></tr>
<tr><td>Perceptions on code smells in Elixir</td><td>(Short overview of the concept of Elixir-Specific code smells)</td></tr>
<tr><td></td><td>∗ How often does such smells occur in the Elixir systems you have worked with?
(1 = it is very rare; 5 = it is very common)
<table>
<tr><th>Smell (in random order)</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th></tr>
<tr><td>Large messages
[link to description and code example]</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td></tr>
<tr><td>Code organization by process
[link to description and code example]</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td></tr>
<tr><td>Large code generation by macros
[link to description and code example]</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td></tr>
<tr><td>"Use" instead of "import"
[link to description and code example]</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td></tr>
<tr><td>Alternative return types
[link to description and code example]</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td></tr>
<tr><td>Dynamic atom creation
[link to description and code example]</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td></tr>
</table></td></tr>
<tr><td></td><td>∗ How relevant are these smells in Elixir systems (evaluated as their potential to have a negative impact on maintainability, comprehensibility, and evolution)?
(1 = very low impact and relevance; 5 = high impact and relevance)
<table>
<tr><th>Smell (in random order)</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th></tr>
<tr><td>Large messages
[link to description and code example]</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td></tr>
<tr><td>Code organization by process
[link to description and code example]</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td></tr>
<tr><td>Large code generation by macros
[link to description and code example]</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td></tr>
<tr><td>"Use" instead of "import"
[link to description and code example]</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td></tr>
<tr><td>Alternative return types
[link to description and code example]</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td></tr>
<tr><td>Dynamic atom creation
[link to description and code example]</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td><td>◎</td></tr>
</table></td></tr>
</table>

∗: *Optional answers (one per line).*

Table A.8: Questionnaire C - Perceptions on Elixir-Specific code smells

Topic	Questions
Perceptions on code smells in Elixir	*(Short overview of the concept of Elixir-Specific code smells)*

∗ **How often does such smells occur in the Elixir systems you have worked with?**

(1 = it is very rare; 5 = it is very common)

Smell (in random order)	1	2	3	4	5
Agent Obsession [link to description and code example]	◎	◎	◎	◎	◎
Complex extractions in clauses [link to description and code example]	◎	◎	◎	◎	◎
Using exceptions for control-flow [link to description and code example]	◎	◎	◎	◎	◎
Complex else clauses in with [link to description and code example]	◎	◎	◎	◎	◎
Accessing non-existent map/struct fields [link to description and code example]	◎	◎	◎	◎	◎
Unnecessary macros [link to description and code example]	◎	◎	◎	◎	◎

∗ **How relevant are these smells in Elixir systems (evaluated as their potential to have a negative impact on maintainability, comprehensibility, and evolution)?**

(1 = very low impact and relevance; 5 = high impact and relevance)

Smell (in random order)	1	2	3	4	5
Agent Obsession [link to description and code example]	◎	◎	◎	◎	◎
Complex extractions in clauses [link to description and code example]	◎	◎	◎	◎	◎
Using exceptions for control-flow [link to description and code example]	◎	◎	◎	◎	◎
Complex else clauses in with [link to description and code example]	◎	◎	◎	◎	◎
Accessing non-existent map/struct fields [link to description and code example]	◎	◎	◎	◎	◎
Unnecessary macros [link to description and code example]	◎	◎	◎	◎	◎

∗: *Optional answers (one per line).*

Table A.9: Questionnaire D - Perceptions on Elixir-Specific code smells

Topic	Questions
Perceptions on code smells in Elixir	*(Short overview of the concept of Elixir-Specific code smells)*
	∗ **How often does such smells occur in the Elixir systems you have worked with?** (1 = it is very rare; 5 = it is very common)

Smell (in random order)	1	2	3	4	5
Unsupervised process [link to description and code example]	◎	◎	◎	◎	◎
Unrelated multi-clause function [link to description and code example]	◎	◎	◎	◎	◎
Compile-time global configuration [link to description and code example]	◎	◎	◎	◎	◎
Complex branching [link to description and code example]	◎	◎	◎	◎	◎
Speculative Assumptions [link to description and code example]	◎	◎	◎	◎	◎

∗ **How relevant are these smells in Elixir systems (evaluated as their potential to have a negative impact on maintainability, comprehensibility, and evolution)?**

(1 = very low impact and relevance; 5 = high impact and relevance)

Smell (in random order)	1	2	3	4	5
Unsupervised process [link to description and code example]	◎	◎	◎	◎	◎
Unrelated multi-clause function [link to description and code example]	◎	◎	◎	◎	◎
Compile-time global configuration [link to description and code example]	◎	◎	◎	◎	◎
Complex branching [link to description and code example]	◎	◎	◎	◎	◎
Speculative Assumptions [link to description and code example]	◎	◎	◎	◎	◎

∗: *Optional answers (one per line).*

Table A.10: Final remarks questions (equals across all questionnaire versions)

Topic	Questions
Final Remarks	∗ **Please add here any comment about our work and survey:** _____
	Do you want to receive the preliminary results of this study? [YES | NO]

∗: *Optional answer.*

Appendix B

Documents of the Survey on Refactorings for Elixir

In this appendix, we present the instruments used to perform the validation of our catalog of refactorings for Elixir (Chapter 4). In Section B.1, we introduced the Free and Enlightened Consent document that should be signed by all participants of our survey before responding to our questionnaire on refactorings for Elixir. In Section B.2, we presented each of the five versions of our questionnaire.

B.1 Free and Enlightened Consent

Title: Refactorings for systems implemented in the Elixir functional language

Evaluation goals: This study aims to understand the main refactoring strategies used in code implemented in the Elixir functional language.

Survey information: We will ask you questions about your demographics, positions held, experience with Elixir, and your perception about the refactoring strategies in Elixir (use frequency and impact on system quality).

The remainder of this document essentially follows the same structure of topics and content as the Free and Enlightened Consent form signed by participants in our Survey on Code Smells in Elixir (see Section A.1).

B.2 Survey Questions

The questions about demographics and final remarks were identical to those in our Survey on Code Smells in Elixir, as shown in Table A.1 and Table A.10, respectively. The questions regarding perceptions on refactoring strategies for Elixir followed a different format, as presented in the following tables.

Table B.1: Questionnaire A - Perceptions on refactorings for Elixir

Topic	Questions (refactorings listed in random order)
Perceptions on refactorings for Elixir	*(Short overview and definition of prevalence and relevance scales)*
	∗ **Assess the *prevalence* and *relevance* level of the refactoring Generalise a process abstraction.** [A description, a link with additional details, and code examples]
	∗ **Assess the *prevalence* and *relevance* level of the refactoring Converts guards to conditionals.** [A description, a link with additional details, and code examples]
	∗ **Assess the *prevalence* and *relevance* level of the refactoring Folding against a function definition.** [A description, a link with additional details, and code examples]
	∗ **Assess the *prevalence* and *relevance* level of the refactoring Moving "with" clauses without pattern matching.** [A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗: *Optional answers (one per line).*

(Continues)

Table B.1: Questionnaire A - Perceptions on refactorings for Elixir (continued)

Topic	Questions (refactorings listed in random order)
Perceptions on refactorings for Elixir	*(Short overview and definition of prevalence and relevance scales)*

∗ **Assess the *prevalence* and *relevance* level of the refactoring List comprehension simplifications.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Remove dead code.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Behaviour inlining.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Widen or narrow definition scope.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Remove unnecessary calls to length/1.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Add or remove a parameter.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗: *Optional answers (one per line).*

(Continues)

Table B.1: Questionnaire A - Perceptions on refactorings for Elixir (continued)

Topic	Questions (refactorings listed in random order)
Perceptions on refactorings for Elixir	*(Short overview and definition of prevalence and relevance scales)*
	∗ **Assess the *prevalence* and *relevance* level of the refactoring Merging multiple definitions.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Static structure reuse.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Replace "Enum" collections with "Stream".**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Remove nested conditional statements in function calls.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Moving a definition.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Splitting a definition.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗: *Optional answers (one per line).*

Table B.2: Questionnaire B - Perceptions on refactorings for Elixir

Topic	Questions (refactorings listed in random order)
Perceptions on refactorings for Elixir	*(Short overview and definition of prevalence and relevance scales)*
	∗ **Assess the *prevalence* and *relevance* level of the refactoring Extract constant.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Merging match expressions into a list pattern.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Add a tag to messages.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Simplifying Ecto schema fields validation.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Equality guard to pattern matching.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Group Case Branches.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗: *Optional answers (one per line).*

(Continues)

Table B.2: Questionnaire B - Perceptions on refactorings for Elixir (continued)

Topic	Questions (refactorings listed in random order)
Perceptions on refactorings for Elixir	*(Short overview and definition of prevalence and relevance scales)*
	∗ **Assess the *prevalence* and *relevance* level of the refactoring Remove redundant last clause in "with".**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Behaviour extraction.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Pipeline for database transactions.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Replace a nested conditional in a "case" statement with guards.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Reducing a boolean equality expression.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Transform to list comprehension.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗: *Optional answers (one per line).*

(Continues)

Table B.2: Questionnaire B - Perceptions on refactorings for Elixir (continued)

Topic	Questions (refactorings listed in random order)
Perceptions on refactorings for Elixir	*(Short overview and definition of prevalence and relevance scales)*

∗ **Assess the *prevalence* and *relevance* level of the refactoring Simplifying checks by using truthness condition.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring From tuple to struct.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Eliminate single branch.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Introduce import.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗: *Optional answers (one per line).*

Table B.3: Questionnaire C - Perceptions on refactorings for Elixir

Topic	Questions (refactorings listed in random order)
Perceptions on refactorings for Elixir	*(Short overview and definition of prevalence and relevance scales)*
	∗ **Assess the *prevalence* and *relevance* level of the refactoring Improving list appending performance.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring From meta to normal function application.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Convert nested conditionals to pipeline.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Introduce pattern matching over a parameter.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Alias expansion.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Add type declarations and contracts.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗: *Optional answers (one per line).*

(Continues)

Table B.3: Questionnaire C - Perceptions on refactorings for Elixir (continued)

Topic	Questions (refactorings listed in random order)
Perceptions on refactorings for Elixir	*(Short overview and definition of prevalence and relevance scales)*

∗ **Assess the *prevalence* and *relevance* level of the refactoring Move expression out of Case.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Introduce overloading.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Transform nested "if" statements into a "cond".**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Modifying keys in a Map.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Remove import attributes.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Inline macro.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗: *Optional answers (one per line).*

(Continues)

Table B.3: Questionnaire C - Perceptions on refactorings for Elixir (continued)

Topic	Questions (refactorings listed in random order)
Perceptions on refactorings for Elixir	*(Short overview and definition of prevalence and relevance scales)*
	∗ **Assess the *prevalence* and *relevance* level of the refactoring Transform "unless" with negated conditions into "if".**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Rename an identifier.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Generalise a function definition.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Nested list functions to comprehension.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗: *Optional answers (one per line).*

Table B.4: Questionnaire D - Perceptions on refactorings for Elixir

Topic	Questions (refactorings listed in random order)
Perceptions on refactorings for Elixir	*(Short overview and definition of prevalence and relevance scales)*

∗ **Assess the *prevalence* and *relevance* level of the refactoring Grouping parameters in tuple.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Extract function.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Splitting a large module.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Pipeline using "with".**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Function clauses to/from case clauses.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Transform "if" statements using pattern matching into a "case".**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗: *Optional answers (one per line).*

(Continues)

Table B.4: Questionnaire D - Perceptions on refactorings for Elixir (continued)

Topic	Questions (refactorings listed in random order)
Perceptions on refactorings for Elixir	*(Short overview and definition of prevalence and relevance scales)*
	∗ **Assess the *prevalence* and *relevance* level of the refactoring Introduce processes.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Replace function call with raw value in a pipeline start.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Remove single pipe.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Closure conversion.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Turning anonymous into local functions.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Struct field access elimination.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗: *Optional answers (one per line).*

(Continues)

Table B.4: Questionnaire D - Perceptions on refactorings for Elixir (continued)

Topic	Questions (refactorings listed in random order)
Perceptions on refactorings for Elixir	*(Short overview and definition of prevalence and relevance scales)*

∗ **Assess the *prevalence* and *relevance* level of the refactoring Register a process.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Inline function.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Struct guard to matching.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Explicit a double boolean negation.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Replace conditional with polymorphism via Protocols.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗: *Optional answers (one per line).*

Table B.5: Questionnaire E - Perceptions on refactorings for Elixir

Topic	Questions (refactorings listed in random order)
Perceptions on refactorings for Elixir	*(Short overview and definition of prevalence and relevance scales)*
	∗ **Assess the *prevalence* and *relevance* level of the refactoring Simplifying guard sequences.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Moving error-handling mechanisms to supervision trees.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Transform a body-recursive function to a tail-recursive.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Introduce Enum.map/2.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Default value for an absent key in a Map.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Replace pipeline with a function.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗: *Optional answers (one per line).*

(Continues)

Table B.5: Questionnaire E - Perceptions on refactorings for Elixir (continued)

Topic	Questions (refactorings listed in random order)
Perceptions on refactorings for Elixir	*(Short overview and definition of prevalence and relevance scales)*
	∗ **Assess the *prevalence* and *relevance* level of the refactoring Simplifying pattern matching with nested structs.**
	[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Temporary variable elimination.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Transforming list appends and subtracts.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Replacing recursion with a higher-level construct.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Defining a subset of a Map.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Extract expressions.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗: *Optional answers (one per line).*

(Continues)

Table B.5: Questionnaire E - Perceptions on refactorings for Elixir (continued)

Topic	Questions (refactorings listed in random order)
Perceptions on refactorings for Elixir	*(Short overview and definition of prevalence and relevance scales)*

∗ **Assess the *prevalence* and *relevance* level of the refactoring Typing parameters and return values.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Remove processes.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Introduce a temporary duplicate definitions.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Reorder parameter.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗ **Assess the *prevalence* and *relevance* level of the refactoring Move file.**

[A description, a link with additional details, and code examples]

Perception	1	2	3	4	5
Prevalence	◎	◎	◎	◎	◎
Relevance	◎	◎	◎	◎	◎

∗: *Optional answers (one per line).*

Appendix C

Refactorings not listed in Chapter 4

To improve the readability of the Chapter 4, we chose not to show the description of all 82 refactorings cataloged by us in Section 4.1.2. However, aiming to make this thesis self-contained, in this appendix we provide descriptions of all refactorings not presented in Chapter 4. More details about each one, including code examples, are available at **`https://github.com/lucasvegi/Elixir-Refactorings`**.

Table C.1: Functional Refactorings compatible with Elixir not listed in Section 4.1.2 (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
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 C.2: Traditional Refactorings compatible with Elixir not listed in Section 4.1.2 (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.

Appendix D

Refactoring Code Smells: Practical Guidelines

In this appendix, we provide a detailed explanation of how the removal of each of the 35 code smells for Elixir can be assisted by refactoring strategies (Chapter 5). Along with listing the refactorings useful for removing each smell and explaining their specific applications, we also document the order in which these refactorings should be performed when they are part of a sequence of operations. Specifically, when a refactoring should be used alone, it is listed with a bullet point (*i.e.,* •) in this appendix. Conversely, when a refactoring forms part of a sequence of operations, it is listed using numbering to define its order (*e.g.,* 1.; 2.; etc.)

This appendix is organized with a section for each code smell, arranged in alphabetical order. This document can **guide developers, especially those beginners to Elixir, on how to systematically remove code smells** and improve the internal quality of their systems implemented with this language.

D.1 Accessing non-existent map/struct fields

- Default value for an absent key in a Map: When trying to access the value of a key from a `Map` dynamically, it is not possible to determine if a key is non-existent or if it has an associated `nil` value. This refactoring can eliminate the smell, as the ambiguity in the returns of dynamic accesses will no longer occur after its application.
- Introduce pattern matching over a parameter: If the smell instance occurs in the function signature, such as in a guard clause, we can use this refactoring to extract the value of that field and then use it in the guard. This way, we can clearly determine if a field does not exist or if it simply has an associated `nil` value.

- Simplifying checks by using truthness condition: Optionally, when using dynamic access to fields of a `Map` and needing to return a default value for non-existent fields or those associated with `nil`, we can perform this refactoring to solve this issue, thus producing cleaner code.

- Explicit a double boolean negation: Optionally, if we are using double boolean negation to check if a dynamically accessed field in a `Map` exists, this refactoring can improve code readability by replacing the unintuitive logic with helper functions that utilize pattern matching.

1. Struct field access elimination: Optionally, if accesses to the same field, whether it exists or not, occur many times within a function, we can use this refactoring to replace these accesses with a temporary variable responsible for storing the value of that field.

2. Equality guard to pattern matching: Optionally, when a temporary variable extracted from a `struct` field is only used in an equality comparison in a guard, extracting and using that variable is unnecessary, as we can perform that equality comparison directly with pattern matching. To do this, we can use this refactoring.

D.2 Agent obsession

1. Generalise a function definition: When functions responsible for directly interacting with the `Agent` are scattered throughout the system, it indicates the presence of this smell. We can refactor them simultaneously using this operation, centralizing the responsibility for interacting with the `Agent` in a single module or function.

2. Moving a definition: After generalizing the functions that were originally responsible for directly accessing the `Agent`, the generic function created to centralize this task might not be located in the most appropriate module. This refactoring can be used to address this issue.

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

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

D.3 Alternative return types

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

D.4 Code organization by process

1. Remove processes: When code unnecessarily uses a process for organizational purposes where a simple module with functions would suffice, this refactoring can be

used to remove the unnecessary concurrent processes and replace them with regular Elixir modules.

2. Remove dead code: When we transform a process into a regular Elixir module, some functions that previously implemented process callbacks (*e.g.,* `GenServer`) may become unnecessary and can therefore be removed using this refactoring.

D.5 Comments

- Extract function: If a comment explains a block of code, that block can be extracted into a separate function. The name of the new function can often be derived from the comment itself.
- Extract expressions: If a comment is intended to explain a complex expression, the expression should be split into understandable sub-expressions using this refactoring.
- Extract constant: If a comment is used to explain magic numbers, this refactoring can replace the comments with constants that have human-friendly names.
- Rename an identifier: If a function, expression, or constant has already been extracted, but comments are still necessary to explain what they do, give a self-explanatory name to them using this operation.
- Typing parameters and return values: If a comment is used to document the types of a function's parameters or the type of its return value, that comment can be replaced by a function specification using the `@spec` module attribute.
- Add type declarations and contracts: Alternatively, if a comment is used to document data structures received as parameters of a function or even returned by a function, this comment can be replaced by a type specification using the `@type` and `@typedoc` module attributes.

D.6 Compile-time global configuration

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

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

D.7 Complex branching

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

D.8 Complex else clauses in with

1. Extract function: When an `else` clause in a `with` statement is used to handle different types of errors that may occur during the execution of the `with` clauses, the code can become confusing. To address this issue, we can use this refactoring to transform expressions in the `with` clauses into separate private functions. Each of these private functions will handle a specific error type, decentralizing the error-handling task.

2. Remove dead code: After extracting the new functions responsible for decentralizing error handling, the `with` statement will no longer need an `else` clause. Therefore, we can use this refactoring to remove the `else` clause.

- Remove redundant last clause in "with": Optionally, if the last clause of the `with` statement used to chain operations is redundant, we can use this refactoring to make the code less verbose and more readable.

- Moving "with" clauses without pattern matching: Optionally, if the `with` statement does not perform pattern matching in the first and/or last clauses, we can use this refactoring to make the code more idiomatic and readable.

D.9 Complex extractions in clauses

- Simplifying pattern matching with nested structs: When using pattern matching to perform deep extraction in nested `structs` passed as parameters to a clause of a multi-clause function, we may create unnecessarily messy and hard-to-understand code. With this refactoring, we can simplify this kind of extraction by performing pattern matching only on the outermost `struct` in the nesting, instead of matching patterns with very internal `structs`.

- Converts guards to conditionals: To prevent complex extractions in clauses, which are used both to access data extracted in guard clauses and in the function body, from making the code confusing, we can use this refactoring to replace all guards with traditional conditionals, consolidating them into a single clause for the function. This way, all extracted data will be used exclusively in the function body.

- Equality guard to pattern matching: Optionally, if an unnecessary temporary variable is extracted from a `struct`'s field in a clause of a multi-clause function and is only used for an equality comparison in a guard, this refactoring can simplify the pattern matching in that clause.
- Struct guard to matching: Optionally, if some clauses of a multi-clause function use guard clauses involving the functions `is_struct/1` or `is_struct/2`, this refactoring can simplify the pattern matching performed by these clauses.
- Remove unnecessary calls to length/1: If a list extraction is performed in a function clause only to use it in an unnecessary call to `length/1` in a guard clause, this extraction can be replaced by using pattern matching directly, eliminating the need for the guard clause. To do this, use this refactoring.
- Function clauses to/from case clauses: When complex extractions are done in clauses of multi-clause functions, making it difficult to understand which data is used inside or outside the function body, we can use this refactoring to transform a multi-clause function into a single clause function. By doing this, we will map the function's clauses into clauses of a `case` statement, ensuring that all extractions occur within the function body.

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

D.10 Data manipulation by Migration

1. Module decomposition (Composite)

    1.1. Splitting a large module: When a module that behaves like `Ecto.Migration` performs both data and structural changes in a database schema, it becomes less cohesive, more difficult to test, and therefore more prone to bugs. In these cases, we should split this module into two, moving only the attributes and functions related to data updates to the new module.

   1.2. Rename an identifier: In some cases, after splitting an original module into several smaller and more cohesive modules, the name of the original module can no longer make sense, providing an opportunity also to perform this refactoring.

2. Extract to outside (Composite)

   2.1. Extract function: We can use this operation in the original module to create a new function responsible for calling the routines for altering the data in the database, now present in the new module.

   2.2. Moving a definition: After extracting this function, we can use this refactoring to reposition it in the new module created after splitting the original module.

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

- Simplifying Ecto schema fields validation: Optionally, we can take the opportunity to apply this refactoring in the original module, making it less prone to errors during the validation of the database schema modified via `Ecto.Migration`.

- Pipeline for database transactions: Optionally, we can also take the opportunity to apply this operation in the new module created only to perform changes on data, thus improving its readability by using `Ecto.Multi`.

D.11 Divergent change

- Module decomposition (Composite)

   1. Splitting a large module: This smell can be removed by creating new cohesive modules and moving related functions into them.

   2. Rename an identifier: In some cases, after splitting an original module into several smaller and more cohesive modules, the name of the original module can no longer make sense, providing an opportunity to also apply this refactoring.

- Moving a definition: If there is already a module `B` that is more suitable to accommodate a function that makes module `A` less cohesive, we can simply move this function from module `A` to module `B` to remove this smell.

- Behaviour extraction: This smell can be removed by creating a new cohesive module `B` and moving related functions of module `A` into it. Thereby, we can use this refactoring to transform module `A` into a *behaviour definition* and create module `B` as a *behaviour implementation* of module `A`, thus achieving this goal.

D.12 Duplicated code

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

the anonymous functions were originally implemented should be updated to use the new local function.

- Merging multiple definitions: There are situations where a codebase may have distinct and complementary functions. Because they are complementary, these functions may have identical code snippets. When identified, these functions can be merged into a new function that will simultaneously perform the processing done by the original functions separately.

- Move expression out of case: When the same expression is repeated at the end of all branches of a `case` statement, this refactoring can be used to eliminate the duplicated code.

- Remove redundant last clause in "with": When the last clause of a `with` statement is composed of a pattern identical to the predefined value to be returned by the `with` in case all checked patterns match, this clause is considered redundant. Therefore this duplicated code can be eliminated using this refactoring.

- Static structure reuse: When identical `tuples` or `lists` are used at different points within a function, they are unnecessarily recreated by Elixir. Use this refactoring to eliminate these redundant recreations by assigning the structures to variables, allowing them to be shared throughout the code.

- Introduce import: When a module `A` calls many functions from a module `B`, the name of module `B` may appear repetitively in the code of module `A` due to fully-qualified name calls. This type of duplicated code can be eliminated by importing module `B` into module `A`.

- Widen or narrow definition scope: When we encounter the same anonymous function defined in different parts of the codebase, these functions should be transformed into a local function, eliminating code duplication by expanding the scope of the original function. This refactoring serves as an alternative to Turning anonymous into local functions.

1. Introduce Enum.map/2: When each element of a `list` is manually generated by repeatedly calling the same function, we can use this refactoring to eliminate duplicated code and make the code more idiomatic.

2. Transform to list comprehension: Optionally, after using the previous refactoring to eliminate duplicated code, we can use this operation to convert the call to `Enum.map/2` into semantically equivalent code that can be also more declarative and easier to read.

3. List comprehension simplifications: Optionally, after using the previous refactoring as part of a sequence of atomic refactorings to eliminate duplicated code, we can also use this refactoring to revert the previous transformation, thereby retaining calls to the higher-order function `Enum.map/2` instead of using list comprehensions.

D.13 Dynamic atom creation

1. Extract function: We can replace a call to the function `String.to_atom/1` with an explicit conversion. To do this, we can use this refactoring on the calls to `String.to_atom/1`, creating a new function. This new function should take a `string` as a parameter and convert it to an `atom`. The body of this function should include a conditional to check the content of the `string`. Depending on its content, the function will return a different `atom` directly.

2. Introduce pattern matching over a parameter: After extracting a new function for explicit conversions from `strings` to `atoms`, we can use this refactoring to transform the function into a multi-clause function, where each clause is responsible for returning one of the possible converted `atoms`.

- Folding against a function definition: If there is already a function in the module responsible for performing the explicit conversion of a `string` to an `atom` (*e.g.,* a function extracted for this purpose at a different point in the same module), we can replace a call to `String.to_atom/1` with a call to that function.

- Gradual change (Composite)

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

D.14 Feature envy

- Extract to outside (Composite)

    1. Extract function: If part of a function calls more functions from other modules than from the module where it is defined, we can use this refactoring to separate the envious part into a new function.
    2. Moving a definition: If a function calls more functions from other modules than from the module where it is defined (*e.g.,* the function extracted in the previous refactoring), we can move it to the module most accessed by it.

- Remove import attributes: By using this refactoring, we can directly identify the origin of a function being called by another function. This way, we can more clearly identify a Feature envy instance, helping us to subsequently remove it.

D.15 GenServer envy

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

D.16 Inappropriate intimacy

1. Closure conversion: A specific type of Inappropriate intimacy can be seen in closures, which are impure anonymous functions, as they access variables outside their scope. One way to remove this smell is by transforming the closure into a pure anonymous function.

2. Add or remove a parameter: Imagine the scenario where an anonymous function `A` is defined within a named function `B`. Furthermore, consider that `A` accesses a variable in its body that is a parameter of `B`, which is not passed as a parameter to `A` (*i.e.,* function `A` is a closure). When applying the previous refactoring to remove the Inappropriate intimacy instance in `A`, we may end up with unused parameters in the named function `B`. Therefore, we can use the present refactoring in `B` to fix it.

- Moving a definition: If a function in module `A` is impure because it accesses internal details of module `B` without receiving them through its parameters, we can remove this smell by moving the function from `A` to `B`, thus reducing the coupling between modules.

- Module decomposition (Composite)

    1. Splitting a large module: If the modules involved in an instance of this smell have common interests, we can use this refactoring to put their commonality in a new module and make them high-cohesive and low-coupling modules.
    2. Rename an identifier: In some cases, after splitting an original module into several smaller and more cohesive modules, the name of the original module can no longer make sense, providing an opportunity to also apply this refactoring.

D.17 Large class

- Module decomposition (Composite)

    1. Splitting a large module: When a module does the work of two or more, it becomes large, poorly cohesive, and difficult to maintain. In these cases, we should split this module into several new ones, moving to each new module only the attributes and functions with purposes related to their respective goals.

2. Rename an identifier: In some cases, after splitting an original module into several smaller and more cohesive modules, the name of the original module can no longer make sense, providing an opportunity to also apply this refactoring.

- Behaviour extraction: This operation is helpful if it's necessary to have a list of operations and behaviors that client modules can reuse, thus reducing their sizes.
- Moving a definition: When a function is defined in module `A` but is more suited to the responsibilities of module `B`, we can move it to `B` to reduce the size of `A`.

D.18 Large code generation by macros

- Extract to outside (Composite)

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

D.19 Large messages

- Defining a subset of a Map: If we are originally sending a complete `Map` from one process to another, but actually only need to send a few fields from this `Map`, we can use this refactoring to help reduce the size of the message sent.
- Extract expressions: When we use `spawn/1` to perform message passing between processes, we can pass an anonymous function as a parameter to `spawn/1` that accesses a `Map` field in its body. This will still copy over all of the `Map`, because the `Map` variable is being captured inside the spawned function. The function then extracts the field, but only after the whole `Map` has been copied over. Suppose we only need to send one field of a `Map` between processes. In that case, we can reduce the size of this

message by using this refactoring to store only the necessary value of the `Map` field in a temporary variable. This variable is then used in the spawned function.

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

D.20 Long function

- Extract function: If a comment is needed to explain some part of the function body, this refactoring can be used to create a new function to be called at the location of this comment, thus decreasing the size of the original function.
- Transform nested "if" statements into a "cond": If a function uses many nested `if` conditionals, this can greatly increase its size. In these cases, we can use this refactoring to decrease the size of the function.
- Folding against a function definition: If a function performs operations that can be delegated to other existing functions, it may become unnecessarily long. In these cases, we can use this refactoring to reduce the size of the original function.
- Remove dead code: If a function contains code that is no longer used, it may become unnecessarily long. In these cases we can use this refactoring to reduce the size of the original function.
- Simplifying checks by using truthness condition: When we know that a given data item can be `nil` and we need to return a default value if it is indeed `nil`, we can use this refactoring to reduce the number of lines in a function while maintaining clean and self-explanatory code.
- Generalise a function definition: When different functions have equivalent expression structures, these equivalent expressions can be generalized into a new higher-order function, thus reducing the size of the original functions.
- Introduce pattern matching over a parameter: When a function has many branches in its body that depend on values received as parameters, it can become unnecessarily long. We can use this refactoring to create short multi-clauses for the original function, where each clause handles a different branch.

- Replace pipeline with a function: When a function is unnecessarily long due to a pipeline of function calls that can be replaced by a single call, we can use this refactoring to address the issue.

- Default value for an absent key in a Map: A function can have its number of lines reduced by using this refactoring to replace the use of `if` statements that check the return of `Map.has_key?/2` with a call to the `Map.get/3` function.

- Defining a subset of a Map: When a function is unnecessarily long because it manually creates a subset of a `Map` by individually accessing each of the desired key/-value pairs, we can use this refactoring to delegate this task to a call to `Map.take/2`.

- Pipeline using "with": When a function uses nested conditionals solely to control a sequence of function calls, it can become long. By using this refactoring, we can make the function more idiomatic and reduce its number of lines of code.

- Remove redundant last clause in "with": This refactoring can reduce the number of lines of code used by a `with` statement and consequently shorten the size of the function that uses this statement.

- Modifying keys in a Map: When a function is unnecessarily long because it manually replaces a `Map` key with a new key using `Map.get/2`, `Map.put/2`, and `Map.delete/2` functions together, we can use this refactoring to delegate this task to a call to `Map.new/2`, thus significantly reducing the volume of lines of code.

- Remove nested conditional statements in function calls: When a function uses unnecessary nested conditional statements, it can become long and less readable. In such circumstances, we can use this refactoring to replace the unnecessary nested conditional statements with less bulky code that maintains the same behavior, thus making the code simpler and more readable.

- Splitting a definition: When the refactoring Merging multiple definitions is used carelessly, there is a risk of creating a long function. In such cases, we can undo this operation using the present refactoring, thereby generating smaller, separate functions.

- Merging match expressions into a list pattern: If a function uses many lines of code with expressions that assign results to temporary variables but can be replaced by a single `list` generated through pattern matching, we can use this refactoring to reduce the size of the function.

1. Convert nested conditionals to pipeline: When a function uses nested conditionals solely to control a sequence of function calls, it can become long. By

using this refactoring, we can give the function a more functional appearance and reduce the number of lines of code.

2. Replace function call with raw value in a pipeline start: When using the previous refactoring to remove this smell, we may also encounter an opportunity to apply the present operation, making the refactored code more idiomatic.

D.21 Long parameter list

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

D.22 Modules with identical names

- Rename an identifier: In Elixir, there is a naming convention for modules that should be followed when implementing libraries. According to this convention, a library should use its own name as a prefix (namespace) for all its module names (*e.g.,* `LibraryName.ModuleName`). When a library does not adhere to this naming convention, it can lead to name conflicts for the library's clients. To remove this smell, we can rename the modules of a library to adapt them to this convention. It is important to be careful when performing this refactoring on already released libraries, as it may cause breaking changes in client code.

- Gradual change (Composite)

    1. Introduce a temporary duplicate definition: As explained in the previous refactoring, when a library does not follow the naming convention for modules, it can lead to name conflicts in their clients. To remove this smell in already released libraries, we can use this present refactoring on the modules of a library, adapting the names of the copies to this convention. Meanwhile, the modules with names outside the convention should be marked as deprecated but not immediately removed, to avoid breaking client code.
    2. Remove dead code: When modules deprecated by the previous refactoring have remained deprecated long enough for clients to adapt to the new naming conventions, we can use this refactoring to permanently eliminate them, thereby removing the risk of name conflicts.

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

D.23 Primitive obsession

- Introduce parameter struct (Composite)

    1. Grouping parameters in tuple: If primitive/basic type values are used in function parameters to inadequately represent more complex real-world abstractions, you can initially apply this refactoring to group them.
    2. From tuple to struct: A `tuple` used to group parameters can eventually be replaced by a `struct`, thus creating a more robust data structure for this purpose.

- Add type declarations and contracts: We can use this refactoring to generate a type specification to replace primitive/basic values.

D.24 Shotgun surgery

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

D.25 Speculative assumptions

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

D.26 Speculative generality

- Inline function: Use this refactoring to get rid of unused functions or even unnecessarily created ones.
- Inline macro: Similarly to the previous refactoring, use this operation to eliminate unused macros or those that were unnecessarily created.
- Add or remove a parameter: Functions with unused parameters should be reviewed using this refactoring.
- Rename an identifier: Functions with abstract names should be renamed using more specific names that reflect their current task, rather than potential future tasks

they might perform.

- Behaviour inlining: Unnecessary delegation/generalization caused by the excessive use of Elixir's `behaviour` can be removed with this refactoring.

- Remove dead code: In general, any code created unnecessarily to support future features that are never implemented can be refactored using this operation.

- Eliminate single branch: When a conditional is created to potentially provide different treatments for different types of data but actually provides the same treatment to all, we can simplify the code by removing unnecessary complexity.

D.27 Switch statements

1. Replace conditional with polymorphism via Protocols: Suppose the same sequence of conditional statements appears duplicated in the code. In that case, we may be forced to make changes in multiple parts of the code whenever a new check needs to be added to these duplicated sequences of conditional statements. This refactoring introduces polymorphism to data structures, thus improving the code's extensibility to handle flow controls based on data types.

2. Converts guards to conditionals: The inverse operation of this present refactoring can be used to complement the previous refactoring operation, combining polymorphism with guard clauses in the implementation of the functions defined in the created `Protocol`.

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

- Introduce overloading: We can overload a function, transforming it into a multi-clause function, to eliminate a conditional.

1. Extract to outside (Composite)

    1.1. Extract function: Often the duplicated switch/conditional statement switches on a "type code". To isolate a switch/conditional in a *type code host*, start by extracting one of the duplicated switch/conditional statements into a new function.

1.2. Moving a definition: Second, the extracted function should be moved to the right module.

2. Generalise a function definition: After the previous composite refactoring, the moved function should be transformed into a higher-order function. One of the parameters of this generalized function will receive a function as an argument, responsible for defining the strategy of the internal conditional check.

3. Folding against a function definition: Finally, all points in the code with duplicated switch/conditional statements can use this refactoring to replace the duplicated code with calls to the previously defined higher-order function.

D.28 Unnecessary macros

- Inline macro: When code is implemented as a `macro` but could be implemented as a conventional function in Elixir, we can use this refactoring to remove this smell, improving the readability of the code.

- Extract to outside (Composite)

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

D.29 Unrelated multi-clause function

1. Rename an identifier: A possible solution to this smell is to use this refactoring to break up the business rules that are mixed into several different simple functions. Specifically, groups of clauses from the original function that share related func-

tionalities can be renamed with an identical name, thus forming a new multi-clause function.

2. Function clauses to/from case clauses: Optionally, after renaming the groups of related clauses, thus creating new multi-clause functions, these multi-clause functions can be transformed into single-clause functions by mapping function clauses into clauses of `case` statements.

- Moving a definition: When unrelated clauses of a multi-clause function differ to the point where they do not make sense in the current module, these clauses can be moved to other modules where they fit better, thus making the code more cohesive.

- Struct guard to matching: Optionally, when some of the clauses of a multi-clause function use a guard clause involving the functions `is_struct/1` or `is_struct/2`, we can use this refactoring to simplify the pattern matching performed by these function clause's signature.

- Equality guard to pattern matching: Optionally, when an unnecessary temporary variable is extracted from a `struct`'s field in a clause of a multi-clause function only to be used in an equality comparison in a guard, we can use this operation to simplify the pattern matching performed by this function clause's signature.

- Simplifying guard sequences: Optionally, when a clause of a multi-clause function contains redundant logical propositions, we can simplify the pattern matching performed by this function clause's signature through the transformation carried out by this refactoring.

- Converts guards to conditionals: Optionally, when what differentiates the clauses of a multi-clause function are only the logical checks performed in their guard clauses, we can use this refactoring to replace all guards with traditional conditionals, creating only one clause for the function.

- Simplifying pattern matching with nested structs: Optionally, when using pattern matching to perform deep extraction in nested `structs` passed as parameters to a clause of a multi-clause function, we may create unnecessarily messy and hard-to-understand code. With this refactoring, we can simplify this kind of extraction by performing pattern matching only on the outermost `struct` in the nesting, instead of matching patterns with very internal `structs`.

- Remove unnecessary calls to length/1: Optionally, when unnecessary calls to the function `length/1` are used in guard clauses to differentiate the pattern matching performed by different clauses of a multi-clause function, we can replace these

calls with direct pattern matching on the parameter list of the function clauses, thereby improving code efficiency.

D.30 Unsupervised process

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

D.31 Untested polymorphic behaviors

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

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

D.32 "Use" instead of "import"

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

2. Remove dead code: After adding the `import` directive, we should remove the `use` directive using this refactoring.

- Alias expansion: Optionally, if the directive used to replace `use` is an `alias` and it is in the multi-alias format (*e.g.,* `alias Foo.Bar.{Baz, Boom}`), we can use this refactoring, thus providing an improvement in code readability and traceability.

- Remove import attributes: Optionally, if after replacing `use` with `import` we identify that a module is excessively importing other modules to the point of impairing readability—making it difficult to identify the origin of the functions it calls—we can perform this refactoring to some unnecessary imports. After that, we will then call the functions from the removed `imports` by their fully-qualified names.

D.33 Using App Configuration for libraries

- Explicit a changed function signature (Composite)

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

D.34 Using exceptions for control-flow

1. Rename an identifier: To prevent a library function from always forcing third-party code to handle an error as an exception, we can initially rename the original function, adding a trailing `!` at the end of its name. In Elixir, there is a convention where a `!` (*i.e., trailing* or *bang*) at the end of a function name indicates that it may raise an exception.

2. Introduce a temporary duplicate definition: After renaming the original function by adding a `!` at the end of its name, we can use this refactoring to duplicate the renamed original function. This copy should have the original function's name, without the `!` at the end. Instead of raising exceptions, it should return data in the `tuple` format (*i.e.,* `{:ok, _}` or `{:error, msg}`). The message provided in the error `tuple` should be the same as the one originally displayed in the exception version.

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

1. Introduce processes: Another possibility for removing this smell is, if the module where the function that raises an exception is not a process (*e.g.,* `GenServer` or `Task`), we can use this refactoring to transform the module into a process.

2. Moving error-handling mechanisms to supervision trees: When a process is using defensive programming (*i.e.,* `try..rescue`) to handle exceptions and even control the execution flow, we can use this refactoring to eliminate this type of handling, transitioning instead to the *"Let it crash"* style.

D.35 Working with invalid data

1. Typing parameters and return values: When a library function does not validate the types of its parameters at its signature, we can at least use this refactoring to document these data. This will help the clients of this function (*i.e.,* third-party code) to protect themselves from potential errors caused by invalid data.

2. Add type declarations and contracts: If recurring data structures are found when documenting functions of a library, these structures can be named using the present refactoring, thus creating new reusable data types and increasing the system's readability.

- Introduce pattern matching over a parameter: Optionally, if a client validates the data passed to a library function call using traditional conditional statements, we can use this refactoring to make the client function more idiomatic while still addressing the smell.

- Struct guard to matching: Optionally, if a guard clause involving the functions `is_struct/1` or `is_struct/2` is used to avoid working with invalid data, we can use this refactoring to simplify the code used to remove this code smell.

- Simplifying guard sequences: Optionally, if a guard clause with redundancies is used to avoid working with invalid data, we can use the present refactoring to also simplify the code used to remove this code smell.

- Converts guards to conditionals: Optionally, we can replace guard clauses used in a multi-clause client function to handle invalid data with traditional conditional statements. By doing so, we can use this refactoring to consolidate all data type validations into a single-clause function.