Noname manuscript No.
(will be inserted by the editor)

Understanding Code Smells in Elixir Functional Language

Lucas Francisco da Matta Vegi · Marco Tulio Valente

Received: date / Accepted: date

Abstract Elixir is a functional programming language created in 2012, whose popularity is growing in the industry. Despite this fact, and to the best of our knowledge, there are few works in the literature focused on studying the internal quality of systems implemented with this language. In a preliminary and previous study, we conducted a grey literature review to provide an initial list of Elixir-specific code smells. Aiming to expand the results of this preliminary study, in this work we use a mixed methodology, based on the interaction with the Elixir developer community and on the mining of issues, commits, pull requests, and the source code in GitHub repositories to prospect and document new code smells for this language. As a result, 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, which also affect Elixir systems. We validated this catalog by conducting a survey with 181 experienced Elixir developers from 37 countries and all continents. In this survey, we assessed the levels of relevance and prevalence of each smell in the catalog. We show that most smells in Elixir have relevance levels capable of impacting the readability, maintainability, and evolution of Elixir systems. Furthermore, most of the smells are not uncommon in production code. Our results have practical implications related to the prevention and removal of code smells in Elixir.

Keywords Code Smells · Elixir · Functional Programming · Grey Literature · Mining Software Repositories · GitHub

Lucas Francisco da Matta Vegi
Department of Computer Science, UFMG, Brazil, E-mail: lucasvegi@dcc.ufmg.br

Marco Tulio Valente
Department of Computer Science, UFMG, Brazil, E-mail: mtov@dcc.ufmg.br

1 Introduction

Elixir is a modern functional programming language. Its name reflects the basis of its inception,[1] since Elixir mixes features of languages such as Ruby, Haskell, Clojure, and Erlang, resulting in an extensible syntax language [35], which allows the development of fault-tolerant systems that are suitable for running in concurrent and distributed environments [1].

As with any programming language, it is natural to expect that Elixir developers make bad design decisions and then implement sub-optimal code structures, known as code smells [13]. These structures decrease the internal software quality, impairing maintainability [42,32] and increasing bug-proneness [18,23].

Although there is vast research on code smells for specific contexts [16, 15,28,10,11,26,29,20,5], it is mostly focused on the object-oriented paradigm. Sobrinho *et al.* [31] 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 [7] is the only work we found about smells in functional languages, but targeting the Haskell programming language.

In contrast to the lack of studies on code smells in functional languages, Elixir is becoming very popular in the industry and is currently being used by companies, such as Discord, Heroku, and PepsiCo.[2] This was a strong motivation for us to explore code smells in a functional language like Elixir.

In a preliminary short paper [37], we reviewed the grey literature to catalog code smells in Elixir, finding discussions about 11 traditional smells and 18 Elixir-specific smells. Therefore, our present work is an extension of this previous study since we used other complementary methodologies for prospecting, documenting, and validating smells. Specifically, we extend the previous study as follows:

1. In our preliminary short paper [37], we limited ourselves to cataloging code smells based on a review of 17 grey literature documents. In the present work, we broadened our scope by identifying code smells from 71 additional documents of two other source types (Interaction with the Elixir community and Mining GitHub Repositories).
2. We documented the code smells found in our previous study [37] in a GitHub public repository [36]. This repository has been promoted on the main Elixir community communication channels in order to encourage the direct interaction of developers with us.
3. As a result of our interaction with the Elixir community, a total of 13 issues and 12 pull requests were submitted to our repository, with 83 comments

[1] Elixir and mixture are synonymous terms in pharmacology. `https://www.collinsdictionary.com/dictionary/english-thesaurus/elixir`

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

being made on them. These contributions allowed us to improve our catalog and to identify four novel code smells.

4. We conducted a mining study where we analyzed 301 artifacts from Elixir code repositories on GitHub. In 46 artifacts we were able to extract code smells for our catalog, including two novel ones.
5. We conducted a survey with 181 Elixir developers to validate the proposed catalog of code smells. 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*).

The contributions of this paper are threefold: (1) we cataloged 23 novel Elixir-specific code smells and categorize them into two different groups (LOW-LEVEL CONCERNS SMELLS and DESIGN-RELATED SMELLS); (2) we find that at least 12 traditional code smells (as proposed by Fowler and Beck [13]) are also present in Elixir systems; (3) We showed through the results of our 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 therefore common in production code. These findings have practical implications for developers and researchers.

The remainder of this paper is organized as follows. In Section 2, we present background information on the Elixir language and code smells. In Section 3, we present our catalog of code smells for Elixir. Also, we detail our research questions, methods, and threats to validity related to prospecting and documenting code smells. In Section 4, we go into depth on the survey we conducted with developers to validate our catalog of Elixir smells. This section presents the survey's findings, threats to validity, and the methods used to design the questionnaires and to reach the respondents. In Section 5, we discuss the implications of our results. Finally, we present related work in Section 6. Conclusions and ideas for future work are discussed in Section 7.

2 Background

2.1 Elixir Language

Elixir is a modern functional programming language that performs well in parallel and distributed environments. Overall, the popularity of functional languages is on the rise in the industry, with a growing number of users [4]. More than 300 companies around the world already use Elixir in production code, some of them are well-known, such as Pinterest, Adobe, and Spotify.

With Elixir, developers can build scalable applications more transparently, without directly worrying about the use of semaphores or other strategies for synchronizing access to resources that are common in *multithreaded* environments [35]. It was conceived in 2012 by José Valim, inspired by a mix of other languages. 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. Code developed in Elixir is interoperable with code developed in Erlang, as they also 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 [1]. In addition, Elixir is a polymorphic and extensible language, as it has inherited resources such as `protocols` and `macros` from Clojure. Elixir is used in Web development,[3] embedded software,[4] machine learning systems,[5] and for many other purposes.

Elixir programs are organized by `modules`, which are groups of functions. Listing 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),[6] which is an intelligent terminal that allows developers not only to run their code but also to test and access its documentation.

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

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

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

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

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

[6] https://hexdocs.pm/iex/1.13/IEx.html

```
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 characteristic operator 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.

2.2 Code Smells

Fowler and Beck [13] coined the terms *code (or bad) smells* to name sub-optimal code structures that can harm software maintenance and evolution [42, 32]. In addition to coining the terms, they cataloged 22 code smells, which are listed in Table 1. There are also other terms that are mostly synonyms of code smells, such as anti-patterns [6], code anomalies [22,21] and bad practices [34].

In addition to decreasing maintainability and hampering evolution, code smells can increase bug-proneness [18,23]. As shown by Fontana *et al.* [12], the impacts on software quality caused by code smells are not homogeneous and may differ among domains. Developers' perception of code smells can also vary across different software contexts [33]. For this reason, code smells for specific contexts like Android [16], iOS [15], JavaScript [28,10,11], and others have also been studied.

Table 1 Traditional code smells

Code smells cataloged by Fowler & Beck [13]	
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

3 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 [39]. In this context, previous research cataloged code smells for specific-contexts [16,15,28,10,26,29], but none directly consider functional languages such as Elixir. 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 [13] are important in the Elixir context. In the present study, we identified nine new discussions about these smells, including one about a previously undiscussed traditional smell in our preliminary short paper [37].

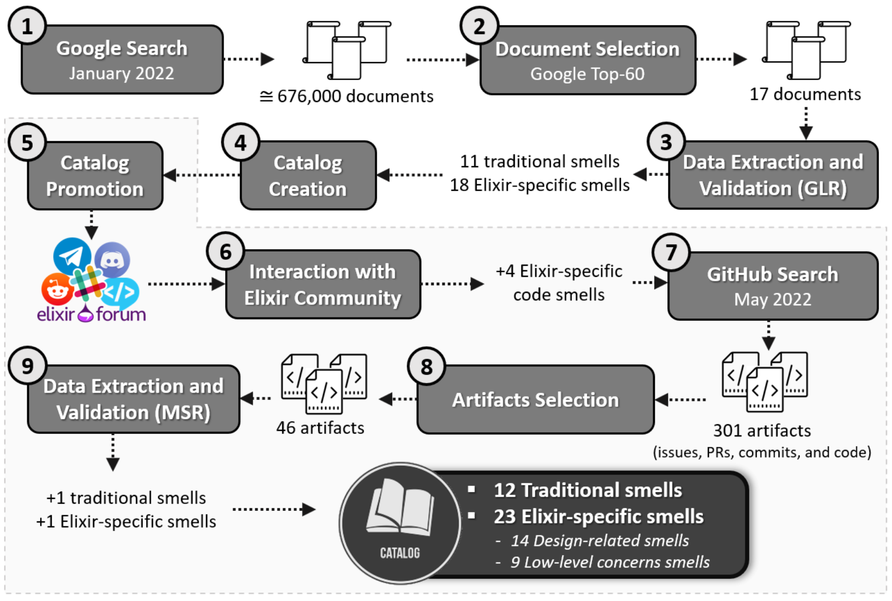
RQ2. *Do Elixir developers discuss other smells?* Next, we investigate discussions about design and code problems specific to Elixir systems, thus referring to them as Elixir-specific smells. In this study, we found 37 new discussions related to Elixir-specific code smells. Among them, there were discussions on five smells that were not previously cataloged in our preliminary short paper [37].

We dedicate Section 3.1 to present the mixed methodology applied to proposing a catalog of code smells for Elixir. In Section 3.2, we present the traditional smells discussed by developers in the Elixir context. Next, in Section 3.3, 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 3.4 discusses threats to validity.

3.1 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 1 summarizes the steps we followed to propose the catalog of code smells for Elixir. The steps that differentiate our present work from our preliminary study [37] are highlighted within the grey dashed polygon. We also detail all these steps in the following paragraphs.



1 Google Search January 2022
≅ 676,000 documents
2 Document Selection Google Top-60
17 documents
3 Data Extraction and Validation (GLR)
11 traditional smells
18 Elixir-specific smells
4 Catalog Creation
5 Catalog Promotion
elixir forum
6 Interaction with Elixir Community
+4 Elixir-specific code smells
7 GitHub Search May 2022
301 artifacts
(issues, PRs, commits, and code)
8 Artifacts Selection
46 artifacts
9 Data Extraction and Validation (MSR)
+1 traditional smells
+1 Elixir-specific smells
CATALOG
12 Traditional smells
23 Elixir-specific smells
14 Design-related smells
9 Low-level concerns smells

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

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 [17,43]. According to Garousi *et al.* [14], 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.

Listing 3 Search query before calibration

```
("Elixir" OR "phoenix framework") AND
("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[7] with the same name and that is not related to our research. Listing 4 presents the search query after this calibration.

Listing 4 Search query after calibration

```
("Elixir") AND
("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.

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.* [14], 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 [24], 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.

The documents were analyzed by the two authors. In particular, our selection criteria were based on an adaptation of the Quality Assessment Checklist proposed by Garousi *et al.* [14]. 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?[8] (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?

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

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

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.

To ensure that these 17 documents also meet at least one of Garousi's Authority of the Producer criteria, we checked the authors' profiles on the platforms where these documents were made available and concluded that all these 17 documents were also valid according to this second selection criterion. The representativeness of the selected documents ranges from documents written by a single developer and then discussed on StackOverflow, Reddit, or Elixir Forum; to documents created by companies, such as Plataformatec[9] or even the Elixir official documentation.

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

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 preference. This same type of disagreement occurred between the authors 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 [13]. 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.

9 `https://plataformatec.com/`

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 2 presents the topics used to document each code smell of our catalog, which is also available in a GitHub public repository [36].

Table 2 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.

Table 3 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 and also by the two authors of this study, seeking to select only those related to internal quality problems and not personal implementation preferences, for example.

As shown in Table 4, 20 documents were accepted by the authors 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,[10] 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 and authors 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 authors 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."

Table 4 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.

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

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.

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.* [8], 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 4). 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 two authors of this work 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 [36]. Third, we adapted Garousi's Authority of the Producer criteria [14] for the context of this MSR study. 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 5 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 [36] or just direct links to it.

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;[11] and from very popular repositories, such as Elixir Language[12] or Phoenix framework.[13]

[11] https://github.com/Finbits

[12] https://github.com/elixir-lang

[13] https://github.com/phoenixframework

Table 5 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

9) Data Extraction and Validation (MSR): After selecting 46 artifacts in the previous step, the first author of this work 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 second author, 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 authors.

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 GitHub, there was no consensus on whether this is a code smell. Also, among the authors of this work, one author 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 author 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 6, most discussions were found in Pull Requests and they refer more to Elixir-specific smells than to traditional smells.

Thus, only one novel Elixir-specific smell emerged from the MSR study. 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 study.

Table 6 Code smell discussions by artifact type (MSR)

Artifact	**Traditional**	**Elixir-specific**
	Discussions (#Novel)	*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)

3.2 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 7. 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 study are started with the letter "M".

Table 7 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

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. [...]"

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 study, 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`[14] 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 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 study. GitHub users who participated in M10's discussions clearly refer to a situation related to it:

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

"[...] 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 [13], this smell can be removed through the use of polymorphism, as also suggested in M10.

By analyzing Table 7, 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 study.

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.

3.3 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). Table 8 summarizes the 14 smells classified as Design-Related, and Table 9 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 [36].

3.3.1 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

Table 8 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

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 Design-Related smell. 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 [...]."

Listing 5 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 5 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 6.

Listing 6 Refactoring of Complex Extractions in Clauses

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

3.3.2 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 9 illustrates an instance of

Table 9 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

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 9 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
```

Another Low-Level Concerns smell, Working with Invalid Data, has been discussed both in the grey literature review and in the MSR study. G5's author describe 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.

3.4 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) studies. 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.* [14], 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 study, 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 paper authors 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 the authors' 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 two authors 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, validates these smells from the perspective of developers.

4 Catalog Validation

To validate the catalog of code smells for Elixir presented in Section 3, we conducted a survey with developers who use this language. Particularly, we 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).

We dedicate Section 4.1 to present the methodology applied to design, conduct, and analyze the results of our survey. In Section 4.2, we present and discuss the results on the prevalence and relevance of code smells in Elixir. Finally, Section 4.3 discusses threats to validity.

4.1 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 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. All the questions[15] that were asked to the participants during the study are available online in the replication package [38] of this work. Particularly, the questions were organized according to the following topics:

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

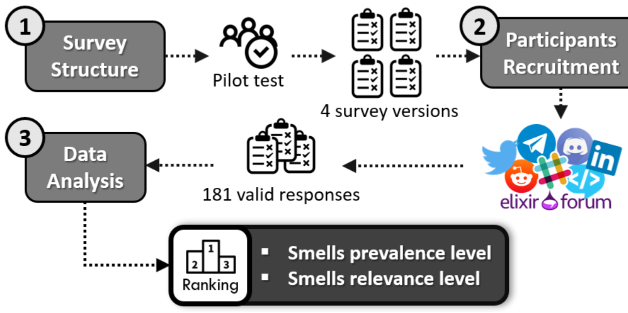


Fig. 2 Overview of survey on code smells in Elixir

- *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 10 presents the distribution of the smells in the survey versions.

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.

Table 10 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.

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 [3], 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,[16] 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, DevTalks, 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. 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, 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 using Elixir.

[16] #MyElixirStatus and #ElixirLang

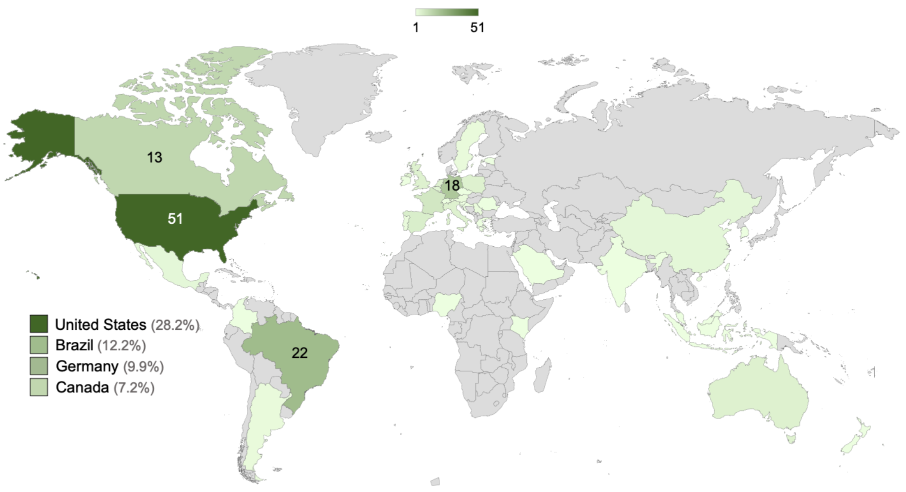


Fig. 3 Countries where the survey participants reside. The legend presents the top 4 countries with most participants

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 11), 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.

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.

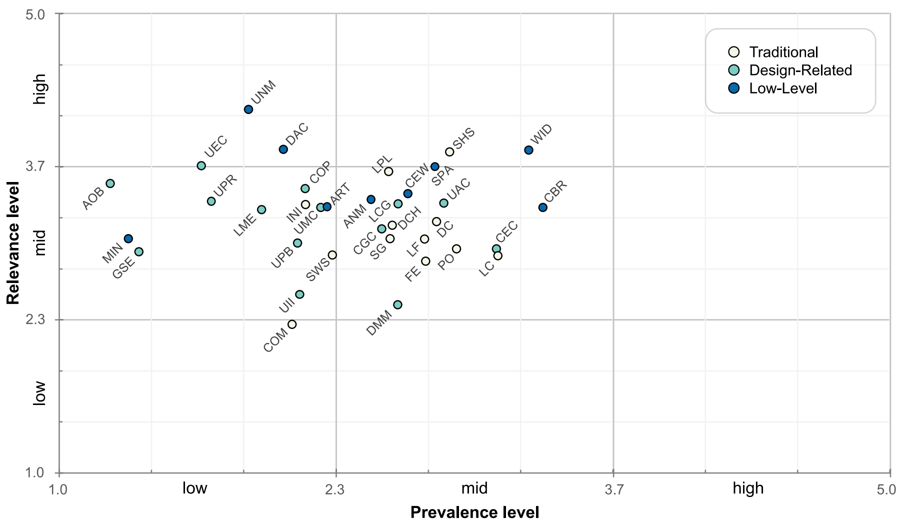
Table 11 Responses for each survey version

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

To assess whether the respondents' level of experience influenced their perception of the prevalence and relevance of smells, we used the Mann-Whitney test [30]. 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,[17] and considered a significance level of 0.05.

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

As can be seen in Figure 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 and 2.3), MID (average scores between 2.3 and 3.7), and HIGH (average scores between 3.7 and 5).



ANM	: Accessing Non-existent Map/Struct Fields	AOB	: Agent Obsession
ART	: Alternative Return Types	CBR	: Complex Branching
CEC	: Complex Extractions in Clauses	CEW	: Complex Else Clauses in With
CGC	: Compile-time Global Configuration	COM	: Comments
COP	: Code Organization by Process	DAC	: Dynamic Atom Creation
DC	: Duplicated Code	DCH	: Divergent Change
DMM	: Data Manipulation by Migration	FE	: Feature Envy
GSE	: GenServer Envy	INI	: Inappropriate Intimacy
LC	: Large Class	LCG	: Large Code Generation by Macros
LF	: Long Function	LME	: Large Messages
LPL	: Long Parameter List	MIN	: Modules with Identical Names
PO	: Primitive Obsession	SG	: Speculative Generality
SHS	: Shotgun Surgery	SPA	: Speculative Assumptions
SWS	: Switch Statements	UEC	: Using Exceptions for Control-flow
UAC	: Using App Configuration for Libraries	UII	: "Use" Instead of "Import"
UMC	: Unrelated Multi-clause Function	UNM	: Unnecessary Macros
UPB	: Untested Polymorphic Behaviors	UPR	: Unsupervised Process
WID	: Working with Invalid Data		

Fig. 4 Developers' perception of code smells in Elixir (RQ3)

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

The scatter plot reveals that the three code smells with the highest levels of relevance are respectively Unnecessary Macros (UNM, 4.16),[18] 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 smell (CBR, 3.33). Its prevalence level is mid, although it is close to the lower limit of the high prevalence zone (see Figure 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.

The level of experience of the developers influenced their perception of just five code smells from our catalog. Specifically, the number of Elixir projects

[18] When discussing each smell, we are adding between parentheses the acronym used in Figure 4 (*e.g.,* UNM) and the respective average score of the survey answers (*e.g.,* 4.16).

the developers worked on affected their perception of four out of the 35 smells, as shown in Table 12. 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 12 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 13, 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.

Table 13 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.

The complete results with the prevalence and relevance levels of each of the 35 code 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 work [38]: `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.

4.3 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.* [20], 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.

5 Implications

Based on our results, 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 [37], only three code smells from our catalog are automatically detected by Credo,[19] 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

[19] `http://credo-ci.org/`

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.

4. Researchers should investigate strategies for refactoring Elixir-Specific smells. As our results indicate, code smells are not rare in Elixir systems, and they can cause non-negligible negative impacts. So, it is important that further studies on code smells in Elixir are carried out. They should focus not only on expanding our catalog but mainly on defining refactoring strategies for the smells in our catalog, as Fowler and Beck [13] have done for traditional smells.

6 Related Work

As code smells are context-sensitive, other studies were carried out to catalog code smells in specific domains. Reimann *et al.* [27] proposed a catalog with 30 Android-specific code smells, extracted from the grey literature. Hecht *et al.* [16] 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.* [15] 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 [10] 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 [11] 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.* [28] 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.* [26] 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.

There are other works that have cataloged code smells in even more specific contexts, such as the configuration management language Puppet [29], the command language Bash [9], and specific smells for video game development [5,20]. In general, as in our work, these papers also use grey literature and investigate real projects through repository mining techniques to catalog code smells from specific contexts.

Just like our work, 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.* [20] 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.

Other studies sought to assess developers' perception of traditional code smells [25,41,2,19,40]. Taibi *et al.* [33] 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, 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 [7] describes 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.

7 Conclusion

This paper expands and validates the first catalog of code smells for Elixir [37]. We used a mixed methodology, based on a grey literature review, interactions with the Elixir community, and mining GitHub code repositories, to catalog 35 code smells that occur in Elixir systems, 23 of which are novel and specific to the language. These Elixir-specific smells were categorized in DESIGN-RELATED SMELLS (14) and LOW-LEVEL CONCERNS SMELLS (9). In addition to the Elixir-specific smells, our catalog also comprises 12 traditional smells, *i.e.,* described by Fowler and Beck [13], which are also found 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. The results showed that most smells have at least MID relevance, and therefore have the potential to impair the readability, maintenance, and evolution of Elixir systems. In addition, the presence of code smells in Elixir systems is not uncommon, therefore requiring the attention of developers, both for preventing the insertion of these sub-optimal code structures and also to conduct their removal when identified.

Future work. As future work, we intend to investigate best practices for removing code smells in Elixir, therefore proposing a catalog of refactoring strategies for Elixir-specific smells. Furthermore, we intend to implement or adapt an existing linter tool, such as Credo, to automatically detect the code smells in our catalog.

Replication Package: We provide the complete dataset used in this paper and a replication package at `https://doi.org/10.5281/zenodo.7430258`. It is composed of (i) documents from grey literature review, all the interactions with the Elixir community, and the artifacts mined from GitHub; (ii) the survey with its respective questions; (iii) the survey results in a fully anonymized format; and (iv) the complete results of the Mann-Whitney test.

Acknowledgements We thank the Elixir community members who spontaneously spread this research and interacted with us on our GitHub repository. We also thank the developers who participated in our survey and shared their perspectives about code quality in Elixir. Finally, José Valim deserves particular thanks for facilitating our communication with the Elixir community and supporting the Research with Elixir initiative (`http://pesquisecomelixir.com.br/`, in Portuguese). This research is supported by a grant from Finbits: `https://www.finbits.com.br`.

Conflict of interest

The authors declare that they have no conflict of interest.

References

1. Almeida, U.: Learn functional programming with Elixir: new foundations for a new world, 1 edn. Pragmatic Bookshelf (2018)
2. Arnaoudova, V., Di Penta, M., 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
3. Baltes, S., Diehl, S.: Worse than spam: Issues in sampling software developers. In: 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 1–6 (2016). DOI https://doi.org/10.1145/2961111.2962628
4. Bordignon, M.D., Silva, R.A.: Mutation operators for concurrent programs in Elixir. In: 21st IEEE Latin-American Test Symposium (LATS), pp. 1–6 (2020). DOI https://doi.org/10.1109/LATS49555.2020.9093675
5. Borrelli, A., Nardone, V., Di Lucca, G.A., Canfora, G., Di Penta, M.: Detecting video game-specific bad smells in Unity projects. In: 17th International Conference on Mining Software Repositories (MSR), p. 198–208 (2020). DOI https://doi.org/10.1145/3379597.3387454
6. Brown, W.J., Malveau, R.C., McCormick, H.W., Mowbray, T.J.: AntiPatterns: refactoring software, architectures, and projects in crisis. John Wiley and Sons (1998)
7. Cowie, J.: Detecting bad smells in haskell. Tech. rep., University of Kent, UK (2005)
8. Dabic, O., Aghajani, E., Bavota, G.: Sampling projects in github for MSR studies. In: 18th IEEE/ACM International Conference on Mining Software Repositories (MSR), pp. 560–564 (2021)
9. Dong, Y., Li, Z., Tian, Y., Sun, C., Godfrey, M.W., 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
10. Fard, A.M., Mesbah, A.: Jsnose: detecting JavaScript code smells. In: 13th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 116–125 (2013). DOI https://doi.org/10.1109/SCAM.2013.6648192

11. Ferreira, F., 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
12. Fontana, F.A., Ferme, V., Marino, A., Walter, B., Martenka, P.: Investigating the impact of code smells on system's quality: an empirical study on systems of different application domains. In: 29th IEEE International Conference on Software Maintenance (ICSM), pp. 260–269 (2013). DOI https://doi.org/10.1109/ICSM.2013.37
13. Fowler, M., Beck, K.: Refactoring: improving the design of existing code, 1 edn. Addison-Wesley (1999)
14. Garousi, V., Felderer, M., 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
15. Habchi, S., Hecht, G., Rouvoy, R., Moha, N.: Code smells in iOS apps: how do they compare to Android? In: 4th IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft), pp. 110–121 (2017). DOI https://doi.org/10.1109/MOBILESoft.2017.11
16. Hecht, G., Benomar, O., Rouvoy, R., Moha, N., Duchien, L.: Tracking the software quality of Android applications along their evolution. In: 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 236–247 (2015). DOI https://doi.org/10.1109/ASE.2015.46
17. Kamei, F., Wiese, I., Lima, C., Polato, I., Nepomuceno, V., Ferreira, W., Ribeiro, M., Pena, C., Cartaxo, B., Pinto, G., 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
18. Li, W., 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
19. Mäntylä, M.V., 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
20. Nardone, V., Muse, B.A., Abidi, M., Khomh, F., Penta, M.D.: Video game bad smells: What they are and how developers perceive them. ACM Trans. Softw. Eng. Methodol. (2022). DOI https://doi.org/10.1145/3563214. Just Accepted
21. Oizumi, W., Garcia, A., da Silva Sousa, L., Cafeo, B., Zhao, Y.: Code anomalies flock together: exploring code anomaly agglomerations for locating design problems. In: 38th International Conference on Software Engineering (ICSE), p. 440–451 (2016). DOI https://doi.org/10.1145/2884781.2884868
22. Oizumi, W.N., Garcia, A.F., Colanzi, T.E., Ferreira, M., von Staa, A.: When code-anomaly agglomerations represent architectural problems? an exploratory study. In: 28th Brazilian Symposium on Software Engineering (SBSE), pp. 91–100 (2014). DOI https://doi.org/10.1109/SBES.2014.18
23. Olbrich, S.M., Cruzes, D.S., 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), pp. 1–10 (2010). DOI https://doi.org/10.1109/ICSM.2010.5609564
24. Page, L., Brin, S., Motwani, R., Winograd, T.: The PageRank citation ranking: bringing order to the Web. Technical report, Stanford InfoLab (1999)
25. Palomba, F., Bavota, G., Penta, M.D., Oliveto, R., Lucia, A.D.: Do they really smell bad? a study on developers' perception of bad code smells. In: IEEE International Conference on Software Maintenance and Evolution (ICSME), p. 101–110 (2014). DOI https://doi.org/10.1109/ICSME.2014.32
26. Punt, L., Visscher, S., 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), pp. 67–77 (2016). DOI https://doi.org/10.1109/ICSME.2016.73
27. Reimann, J., Brylski, M., Aßmann, U.: A tool-supported quality smell catalogue for Android developers. In: Modellbasierte und modellgetriebene Softwaremodernisierung (MMSM), p. 1–2 (2014)

28. Saboury, A., Musavi, P., Khomh, F., Antoniol, G.: An empirical study of code smells in JavaScript projects. In: 24th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 294–305 (2017). DOI https://doi.org/10.1109/SANER.2017.7884630
29. Sharma, T., Fragkoulis, M., Spinellis, D.: Does your configuration code smell? In: 13th IEEE/ACM Working Conference on Mining Software Repositories (MSR), pp. 189–200 (2016)
30. Siegel, S., Castellan, J.: Nonparametric Statistics for the Behavioral Sciences, 2 edn. McGraw-Hill International Editions (1988)
31. Sobrinho, E.V.d.P., De Lucia, A., Maia, M.d.A.: 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
32. Soh, Z., Yamashita, A., Khomh, F., 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), pp. 393–402 (2016). DOI https://doi.org/10.1109/SANER.2016.103
33. Taibi, D., Janes, A., 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
34. Taibi, D., 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
35. Thomas, D.: Programming Elixir |> 1.6: functional |> concurrent |> pragmatic |> fun, 1 edn. Pragmatic Bookshelf (2018)
36. Vegi, L.F.M., Valente, M.T.: Catalog of Elixir-specific code smells (2022). URL `https://github.com/lucasvegi/Elixir-Code-Smells`
37. Vegi, L.F.M., Valente, M.T.: Code smells in Elixir: early results from a grey literature review. In: 30th International Conference on Program Comprehension (ICPC), pp. 1–5 (2022). DOI https://doi.org/10.1145/3524610.3527881
38. Vegi, L.F.M., Valente, M.T.: Understanding code smells in Elixir functional language - Replication Package (2022). URL `https://doi.org/10.5281/zenodo.7430258`
39. Wohlin, C., Runeson, P., Hst, M., Ohlsson, M.C., Regnell, B., Wessln, A.: Experimentation in Software Engineering. Springer (2012)
40. Yamashita, A., Moonen, L.: Do code smells reflect important maintainability aspects? In: 28th IEEE International Conference on Software Maintenance (ICSM), pp. 306–315 (2012). DOI https://doi.org/10.1109/ICSM.2012.6405287
41. Yamashita, A., Moonen, L.: Do developers care about code smells? an exploratory survey. In: 20th Working Conference on Reverse Engineering (WCRE), pp. 242–251 (2013). DOI https://doi.org/10.1109/WCRE.2013.6671299
42. Yamashita, A., 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
43. Zhang, H., Zhou, X., Huang, X., Huang, H., 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), p. 1422–1434 (2020). DOI https://doi.org/10.1145/3377811.3380336