Code Smells in Elixir: Early Results from a Grey Literature Review

Lucas Francisco da Matta Vegi
Marco Tulio Valente
lucasvegi@dcc.ufmg.br
mtov@dcc.ufmg.br
Federal University of Minas Gerais (UFMG)
Belo Horizonte, Minas Gerais, Brazil

ABSTRACT

Elixir is a new functional programming language whose popularity is rising in the industry. However, there are few works in the literature focused on studying the internal quality of systems implemented in this language. Particularly, to the best of our knowledge, there is currently no catalog of code smells for Elixir. Therefore, in this paper, through a grey literature review, we investigate whether Elixir developers discuss code smells. Our preliminary results indicate that 11 of the 22 traditional code smells cataloged by Fowler and Beck are discussed by Elixir developers. We also propose a list of 18 new smells specific for Elixir systems and investigate whether these smells are currently identified by Credo, a well-known static code analysis tool for Elixir. We conclude that only two traditional code smells and one Elixir-specific code smell are automatically detected by this tool. Thus, these early results represent an opportunity for extending tools such as Credo to detect code smells and then contribute to improving the internal quality of Elixir systems.

CCS CONCEPTS

• **Software and its engineering** → **Designing software**; *Software development techniques.*

KEYWORDS

Code Smells, Elixir, Functional Programming

1 INTRODUCTION

Elixir is a modern functional programming language that has high performance in parallel and distributed environments. The language has an extensible and friendly syntax, similar to the Ruby language. With Elixir, developers can create scalable code transparently without worrying so much about synchronization problems

in multithreaded environments [19]. Currently, more than 600 companies around the world—including Pinterest, Discord, Adobe, and Spotify—are already using Elixir in their production code.[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 [4]. These structures decrease the internal software quality, impairing maintainability [20][17] and increasing bug-proneness [9][12].

However, to the best of our knowledge, there are no papers in the literature focused on studying the internal quality of Elixir systems or, in more general terms, of systems implemented in functional languages. For example, Sobrinho et al. [16] conducted a systematic review of articles on code smells published between 1990 and 2017. None of the 104 smells reviewed in their paper refer to functional languages. We also performed a Google Scholar search and did not find more recent code smells papers (2018-2022) that directly consider functional languages such as Elixir.

In order to fill this gap, in this paper, we investigate whether Elixir developers have discussions on code smells. Since Elixir is a new language, we used in our research a grey literature review. This methodology was chosen because it is a good source of information about emerging technologies [8][21].

Our contributions are threefold: (1) we find that half of the traditional code smells also generate discussions among Elixir developers; (2) we catalog 18 novel Elixir-specific code smells and classify them into two different groups (*design-related smells* and *low-level concerns smells*); and (3) we report that Credo,[2] the main linter tool for Elixir, is able to identify only three code smells out of 40 smells, considering traditional and novel ones. This finding motivates the implementation of new code smell detection tools in future work.

The remainder of this paper is organized as follows. In Section II, we present background information on Elixir and code smells. In Section III, we present our research questions and methods. We detail and discuss our early results in Section IV. In Section V, we list threats to validity. Finally, we present related work in Section VI and ideas for future work in Section VII.

2 BACKGROUND

2.1 Elixir

Elixir programs are composed by `modules`, which are groups of functions. Next, we show a simple Elixir module (`Circle`) composed by two functions—area and `circumference`. In lines 10-11, we also show two calls of these functions, using Elixir's interactive shell (IEx).[3]

---

[1]https://elixir-companies.com/en
[2]http://credo-ci.org/
[3]https://hexdocs.pm/iex/1.13/IEx.html

```
1   defmodule Circle do
2     def area(radius) do
3       3.14 * (radius * radius)
4     end
5     def circumference(radius) do
6       2 * 3.14 * radius
7     end
8   end
9   ...
10  iex(1)> Circle.area(15)          # 706.5
11  iex(2)> Circle.circumference(15) # 94.2
```

Modules can also define `structs`, which are key-value pairs similar to objects. We show next an Elixir module with a struct that represents a point in a cartesian plane. This struct has two fields—x and y—that are initialized with `nil` (line 2). In Elixir, a struct has the same name of the module where it is defined . Besides a struct, `Point` has two functions—`distance` and `move`. A pipe operator (`|>`) is used in the `distance` function (line 5) to express function composition. For example, `f() |> g(p)` is equivalent to `g(f(), p)`. Finally, it is important to mention that structs are immutable data structures. For this reason, the `move` function creates a new `Point` with the new location, instead of changing the coordinates of the current `Point` (line 8). Specifically, in Elixir, `%T{...}` is similar to a `new T(...)` in a mainstream object-oriented language.

```
1   defmodule Point do
2     defstruct [x: nil, y: nil]
3     def distance(p1, p2) do
4       Float.pow(p2.x - p1.x, 2) + Float.pow(p2.y - p1.y, 2)
5       |> Float.pow(0.5)
6     end
7     def move(p, delta_x, delta_y) do
8       %Point{x: p.x + delta_x, y: p.y + delta_y}
9     end
10  end
11  ...
12  iex(1)> p1 = %Point{x: 2.0, y: -3.0} #struct creation
13  iex(2)> p2 = %Point{x: 4.0, y: 5.0}  #struct creation
14  iex(3)> Point.distance(p1, p2)  # 8.246211251235321
15  iex(4)> Point.move(p1, 5, 4)    # %Point{x: 7.0, y: 1.0}
```

Elixir programs are executed by BEAM, which is the virtual machine designed to run Erlang programs.[4] BEAM is known to be a fault-tolerant and powerful environment to run distributed systems [1]. Therefore, Elixir programs can easily interoperate with Erlang systems.

2.2 Code Smells

Fowler and Beck [4] coined the terms *code (or bad) smells* to name low quality code structures that can compromise software maintenance and evolution. In the literature, other names are also used with the same purpose, such as anti-patterns [2], code anomalies [11], and bad practices [18].

We performed an initial analysis of the 22 code smells proposed by Fowler and Beck. Although they have been proposed in the nineties, for object-oriented languages, such as Java, we concluded that 20 smells (out of 22) also apply to Elixir. Only *Parallel Inheritance Hierarchies* and *Refused Bequest* are not compatible with Elixir structures and semantics. The reason is that Elixir does not have

---

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

inheritance and these smells are tightly related to this mechanism. On the other side, smells such as *Duplicated Code*, *Long Function*, *Feature Envy*, *Shotgun Surgery*, and others can be directly translated to Elixir code.

However, this initial analysis is not grounded in the practice. Therefore, we cannot affirm that Elixir developers are aware and concerned by the code smells originally proposed by Fowler and Beck. Furthermore, Elixir might have its own code smells, which are specific to the language syntax, semantics, and paradigm.

3 METHODOLOGY

Since Elixir is a new programming language, we have few scientific articles investigating software engineering and quality aspects of Elixir systems. For this reason, the grey literature—composed of blogs, forums, videos, etc—is an interesting source of information for our goals in this study [8][21]. Therefore, we decided to conduct a grey literature review to answer three key research questions:

- **RQ1:** Do Elixir developers discuss traditional code smells?
- **RQ2:** Do Elixir developers discuss other smells?
- **RQ3:** Does a well-known static code analysis tool for Elixir detect code smells?

Figure 1 summarizes the steps we followed in the grey literature review. We also detail these steps in the following paragraphs.
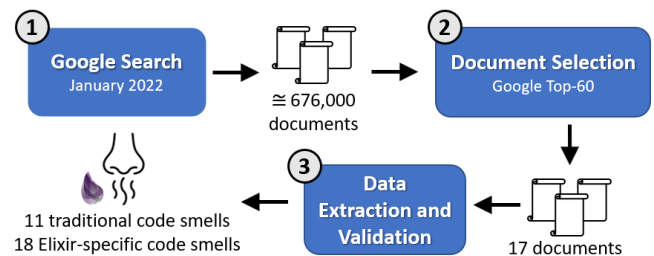

Figure 1: Overview of the grey literature methods

1) Google Search: According to Garousi et al. [5], when defining the keywords for a Google search it is important to perform preliminary searches to calibrate the queries, in order to combine synonyms or to exclude specific terms that might affect the results. After this initial calibration, we define the following search query:

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

This query includes possible code smells synonyms, both in singular and in 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, so it is necessary to limit the number of documents to be analyzed [5]. Hence, we decided to select the top-60 documents returned by Google, i.e., the

first six pages of results. These documents were read by the two authors in order to select only those that are indeed related to our research questions. After this step, 17 documents were selected, which we refer to as D1 to D17. The complete list of documents is available at: https://doi.org/10.5281/zenodo.6025010

3) Data Extraction and Validation: The documents selected in the previous step were analyzed in detail by the first author, in order to retrieve sentences discussing traditional or novel code smells. The extracted sentences were then validated by the second author. Only 2 out of 17 documents—D4 and D15—did not have code smells extracted due to a lack of agreement between the authors.

4 RESULTS

4.1 Do Elixir developers discuss traditional code smells?

In the grey literature, we found discussions about 11 traditional code smells, as shown in Table 1.

Table 1: Traditional code smells (grey literature)

Code Smell in Elixir	Documents	#
Comments	D1, D10, D12, D14	4
Long Parameter List	D1, D16	2
Feature Envy	D1, D6	2
Shotgun Surgery	D1, D17	2
Duplicated Code	D1	1
Long Function	D1	1
Large Class	D1	1
Inappropriate Intimacy	D1	1
Divergent Change	D1	1
Speculative Generalization	D1	1
Primitive Obsession	D3	1

The most discussed traditional smell is *Comments* (4 documents). In D1, for example, the author argues that using comments to document code in Elixir is a bad practice:

"Some people argue that any comment is a code smell, but this is not true. This is about comments that explain what the code does. [Instead] in Elixir we have a strong relationship with code documentation by using doctest. The documentation will describe how to use [a function], bring examples of use [...]. We can access all this via terminal."

For D1's author, if the programmer's objective is to explain the purpose of a function (including the purpose of its parameters, return value, test samples, and so on), this should be done via doctest in Elixir. When doctest is used for code documentation, it is possible to access all the documentation quickly using Elixir's interactive shell (IEx) helper.[5] Also, the usage examples provided for a function can be automatically executed as unit tests, something that is not possible using source code comments.

Although D17's author did not explicitly use the name *Shotgun Surgery* in a document about implementing microservices in Elixir, he clearly refers to situations where particular code modifications require many small changes in different files:

---
[5]https://hexdocs.pm/iex/1.13/IEx.Helpers.html

" [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."

We also found discussions about other smells, such as *Long Parameters List* (in D16):

"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."

By analyzing Table 1, we can also conclude that eight (out of 17) documents have discussions on traditional code smells. However, a single document (D1) concentrates most discussions. This document discusses all, except one, smells detected in our review. Moreover, six (out of 11) smells are only discussed in this specific document.

4.2 Do Elixir developers discuss other smells?

In the grey literature, we found discussions about 18 Elixir-specific code smells, which are summarized in Table 2. The most discussed one is *Exceptions for control-flow* (2 documents). All other Elixir-specific smells are discussed only once each.

To facilitate their discussion, we classify these smells into two different groups: *design-related smells* (10 smells) and *low-level concerns smells* (8 smells). Basically, we classify as *design-related* smells that are more complex, that affect a coarse-grained code element, and that are more difficult to detect. On the other hand, *low-level concerns smells* are more simple and affect a small part of the code.

For example, according to D10's author, although multi-clause functions in Elixir is a powerful resource to keep the code interface small and organized, the abuse of this resource can make the code difficult to understand, characterizing the *Complex multi-clause function* design-related smell, described 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.[...]"

As another example, the low-level concern smell called *Working with invalid data* is described in D5 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.[...]"

By analyzing Table 2, we can draw that eight (out of 17) documents discuss Elixir-specific smells. Only one document (D10) discusses both traditional smells and Elixir-specific smells simultaneously. Furthermore, D5 concentrates the highest number of discussions about Elixir-specific smells, since half of these new smells are detected in this document.

Table 2: Elixir-specific code smells (grey literature)

	Smell	Description	Docs
Design-related smells	GenServer Envy	Using a Task or Agent but handling them like GenServers	D8
	Agent Obsession	When the responsibility for interacting with an Agent process is spread across the system	D8
	Unsupervised process	Library that creates process outside a supervision tree, not allowing users to fully control their apps	D5
	Large messages between processes	Processes that exchange long messages frequently	D13
	Complex multi-clause function	Function with many guard clauses and pattern matching	D10
	Complex API error handling	Function that handles a large number of error types returned by an API endpoint, making it complex	D2
	Exceptions for control-flow	A library that forces clients to handle control-flow exceptions	D5, D11
	Untested polymorphic behavior	Function with a generic Protocol type parameter, but that does not have guards verifying its behavior	D7
	Code organization by process	Library unnecessarily organized as a process, instead as modules and functions	D5
	Data manipulation by migration	Low cohesive module that performs both data and structural changes in a DB schema via Ecto.Migration	D9
Low-level concerns smells	Working with invalid data	Function that does not validate its parameters and therefore can produce non-predicted behaviors	D5
	Map/struct dynamic access	Function that dynamically accesses values from a non-existent field in a struct or map	D7
	Unplanned value extraction	Function that does not force a crash when an incorrect value is extracted from a URL query string	D7
	Modules with identical names	Modules with conflicting names, preventing their simultaneous load	D5
	Unnecessary macro	Using macros instead of functions and structs	D5
	App configuration for code libs	Function that uses global configuration mechanisms, preventing clients from reusing it flexibly	D5
	Compile-time app configuration	Function that uses module attributes to configure itself, therefore preventing run-time configurations	D5
	Dependency with "use" when an "import" is enough	Module that requires knowledge of its internal dependencies to understand its behavior	D5

Note: more details on Elixir-specific code smells are available at https://github.com/lucasvegi/Elixir-Code-Smells

4.3 Does a well-known static code analysis tool for Elixir detect code smells?

Code smell detection tools—including mainstream tools such as SonarQube[6] and research-oriented tools such as DECOR [10]—do not provide support to Elixir. Therefore, we decided to check whether the smells reported in RQ1 and RQ2 are detected by Credo, which is currently the most popular static code analysis tool for Elixir. Credo performs the following checks: software design issues, code readability, refactoring opportunities, warnings, and consistency.

We carefully analyzed Credo documentation (version 1.6.2), aiming to find warnings for the code smells unveiled in our research. Interestingly, most smells listed in RQ1 are not detected by Credo. In fact, only two traditional code smells—*Duplicated code* and *Long Parameter List*—are detected by this tool. Moreover, Credo also does not detect the more complex and design-related smells described in RQ2. Particularly, we concluded that only one Elixir-specific smell—*Compile-time app configuration*—is detected by Credo. This finding represents an opportunity for extending tools such as Credo to detect code smells.

5 THREATS TO VALIDITY

Since our methodology is based on a grey literature review, we analyzed documents that are not peer-reviewed. However, to reinforce the validity of our results, we carefully selected the documents returned by the search engine. Both paper authors did a preliminary reading of all the top-60 documents returned by Google, selecting for analysis only those that are relevant and are written by professionals who work with Elixir.

Another threat concerns the format of the search query. In a grey literature review, there is a risk that important results will not be found due to missing keyword combinations in the search query. To mitigate this threat, as proposed by Garousi et al. [5], we performed preliminary searches, adding and deleting some keywords.

The final threat relates to how we analyzed the smells detected by Credo. However, to minimize this threat, we have thoroughly reviewed all Credo documentation. Although this documentation is rich in details and examples, we might have missed some smells.

6 RELATED WORK

To the best of our knowledge, this is the first work that catalog code smells to a modern functional programming language, more specifically for Elixir. As code smells are context-sensitive, other studies have cataloged code smells for specific contexts, such as Android [7], iOS [6], JavaScript [14][3], CSS [13], and Puppet [15].

As in our work, these papers also used grey literature as a source to discover context-specific code smells. In contrast, they also search for smells in real projects, which we plan to conduct in the near future, as described next.

7 FUTURE WORK

As future work, we plan to extend and validate our list of smells by mining and analyzing issues, pull requests and commits of Elixir open-source projects. We also plan to conduct surveys and interviews with Elixir developers. Next, we plan to invest on tools for detecting smells in Elixir, possibly by extending current static analysis tools, such as Credo. Finally, we have plans to investigate code smells in other modern functional programming languages, such as Clojure.

---

[6]https://www.sonarqube.org/

REFERENCES

[1] Ulisses Almeida. 2018. *Learn functional programming with Elixir: new foundations for a new world* (1 ed.). Pragmatic Bookshelf.

[2] William J. Brown, Raphael C. Malveau, Hays W. McCormick, and Thomas J. Mowbray. 1998. *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley and Sons.

[3] Amin Milani Fard and Ali Mesbah. 2013. JSNOSE: detecting JavaScript code smells. In *13th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 116–125.

[4] Martin Fowler and Kent Beck. 1999. *Refactoring: improving the design of existing code* (1 ed.). Addison-Wesley.

[5] Vahid Garousi, Michael Felderer, and Mika V. Mäntylä. 2019. Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. *Information and Software Technology* 106, 1 (2019), 101–121.

[6] Sarra Habchi, Geoffrey Hecht, Romain Rouvoy, and Naouel Moha. 2017. Code Smells in iOS apps: how do they compare to Android?. In *4th IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. 110–121.

[7] Geoffrey Hecht, Omar Benomar, Romain Rouvoy, Naouel Moha, and Laurence Duchien. 2015. Tracking the software quality of Android applications along their evolution. In *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 236–247.

[8] Fernando Kamei, Igor Wiese, Crescencio Lima, Ivanilton Polato, Vilmar Nepomuceno, Waldemar Ferreira, Márcio Ribeiro, Carolline Pena, Bruno Cartaxo, Gustavo Pinto, and Sérgio Soares. 2021. Grey literature in software engineering: a critical review. *Information and Software Technology* 138, 1 (2021), 1–26.

[9] Wei Li and Raed Shatnawi. 2007. 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 (2007), 1120–1128.

[10] Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and Anne-Francoise Le Meur. 2010. DECOR: a method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering* 36, 1 (2010), 20–36.

[11] Willian Nalepa Oizumi, Alessandro Fabricio Garcia, Thelma Elita Colanzi, Manuele Ferreira, and Arndt von Staa. 2014. When code-anomaly agglomerations represent architectural problems? an exploratory study. In *28th Brazilian Symposium on Software Engineering (SBSE)*. 91–100.

[12] Steffen M. Olbrich, Daniela S. Cruzes, and Dag I.K. Sjøberg. 2010. 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)*. 1–10.

[13] Leonard Punt, Sjoerd Visscher, and Vadim Zaytsev. 2016. 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)*. 67–77.

[14] Amir Saboury, Pooya Musavi, Foutse Khomh, and Giulio Antoniol. 2017. An empirical study of code smells in JavaScript projects. In *24th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 294–305.

[15] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. 2016. Does your configuration code smell?. In *13th IEEE/ACM Working Conference on Mining Software Repositories (MSR)*. 189–200.

[16] Elder Vicente de Paulo Sobrinho, Andrea De Lucia, and Marcelo de Almeida Maia. 2021. A systematic literature review on bad smells–5 w's: which, when, what, who, where. *IEEE Transactions on Software Engineering* 47, 1 (2021), 17–66.

[17] Zéphyrin Soh, Aiko Yamashita, Foutse Khomh, and Yann-Gaël Guéhéneuc. 2016. Do code smells impact the effort of different maintenance programming activities?. In *23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 393–402.

[18] Davide Taibi and Valentina Lenarduzzi. 2018. On the definition of microservice bad smells. *IEEE Software* 35, 3 (2018), 56–62.

[19] Dave Thomas. 2018. *Programming Elixir |> 1.6: functional |> concurrent |> pragmatic |> fun* (1 ed.). Pragmatic Bookshelf.

[20] Aiko Yamashita and Leon Moonen. 2013. To what extent can maintenance problems be predicted by code smell detection? an empirical study. *Information and Software Technology* 55, 12 (2013), 2223–2242.

[21] He Zhang, Xin Zhou, Xin Huang, Huang Huang, and Muhammad Ali Babar. 2020. An evidence-based inquiry into the use of grey literature in software engineering. In *42nd ACM/IEEE International Conference on Software Engineering (ICSE)*. 1422–1434.