Code Smells: A Comprehensive Online Catalog and Taxonomy

Marcel Jerzyk and Lech Madeyski

Abstract Context: Code Smells—a concept not fully understood among programmers, crucial to the code quality, and yet unstandardized in the scientific literature.
Objective: Goal (#1): To provide a widely accessible Catalog that can perform useful functions both for researchers as a unified data system, allowing immediate information extraction, and for programmers as a knowledge base. Goal (#2): To identify all possible concepts characterized as Code Smells and possible controversies. Goal (#3): To characterize the Code Smells by assigning them appropriate characteristics.
Method: We performed a combined search of formally published literature and grey material strictly on Code Smell and related concepts where it might never have been mentioned, along with the term "Code Smell" as a keyword. The results were analyzed and interpreted using the knowledge gathered, classified, and verified for internal consistency.
Results: We identified 56 Code Smells, of which 15 are original propositions, along with an online catalog. Each smell was classified according to taxonomy, synonyms, type of problem it causes, relations, etc. In addition, we have found and listed 22 different types of Bad Smells called hierarchies and drew attention to the vague distinction between the Bad Smell concepts and Antipatterns.
Conclusion: This work has the potential to raise awareness of how widespread and valuable the concept of Code Smells within the industry is and fill the gaps in the existing scientific literature. It will allow further research to be carried out consciously because access to the accumulated information resource is no longer hidden or difficult. Unified data will allow for better reproducibility of the research, and the subsequent results obtained may be more definitive.

Key words: bad smells, code smells, taxonomy, catalog, smell hierarchies

Marcel Jerzyk
Wroclaw University of Science and Technology e-mail: marcerzyk@gmail.com

Lech Madeyski
Wroclaw University of Science and Technology e-mail: lech.madeyski@pwr.edu.pl

1 Introduction and Motivation

The number of new developers increases proportionally to market demand in the IT industry. One of the main issues in software development is technical debt. It is safe to assume that without the assistance and expertise of an experienced developer, the code developed by newcomers is very error-prone. Moreover, this is not the only group that can create code with which problems may arise somewhere in the future. Some practices and tools that try to minimize this problem are Code Reviews, Linters, or Static Analysis Tools. However, despite the best attempts and willingness, knowledge and skills, planning, or design, something that Fowler defined as a "bad code smell" may appear at some point of implementation.

Code Smell is an indication that usually, but not always, corresponds to a deeper problem in the system's architecture, the structure of the project, or the quality of the code, in general. If they are overlooked and left unsolved, they contribute directly to technical debt. It is critical for any successful long-term project to avoid them, but this task is often tricky, ambiguous, and unexpected. Like a Schrodinger cat, one can often only become aware of a problem when one first notices the problem during some further development.

As experience shows, long and semi-long projects sometimes can reach a lifespan counted in years. Thus, it is crucial to be aware of code smells and remove them as soon as one realizes that something might be wrong during the initialization or the development phase. The longer the smelly place remains unrefactored, the more influential and irritating it may become. This situation is not just something that will cause potential employees to have slightly lower morale. The destructive scope is much broader: the code, filled with smells, becomes progressively more challenging to maintain, which will require unnecessarily more dedicated person-hours. Adding more functionalities may take an irrationally long time, and it becomes difficult to estimate how long it would take. Finally, the code can become so complicated that it cannot be further supported.

Since the day it was defined by Fowler, many activities have been carried out from the scientific as well as a practical-lecture side, to name a couple: some new books were created with new Code Smells proposals along with their respective appropriate refactoring methods, new scientific articles trying to impose a more defined framework (definitions, predictors, impact), detection using machine learning techniques.

Despite that, there is a noticeable problem with standardization and access to data and content on this topic. This can be seen in the disproportionate amount of research conducted on individual Code Smells, the use of different, intermixed taxonomies, and different names for the same concepts. One of the latest meta-analyses reached the same conclusions [1]. Despite the confirmed features that characterize it as significant for the quality of the code, the Code Smells has far too little awareness among developers, which it did not deserve.

Both of these things may result directly from the lack of a homogeneous source that could be used for standardized and updated information. In an ideal world, there would be a catalog in which anyone interested could find information on the subject

in an easily digestible form, while allowing the scientist to perform unified research on it. Additionally, anyone could easily update this catalog without the required technical knowledge of web-building, focusing on the merits.

In this paper, we provide such a tool and a summary of the literature available on this topic in terms of definitions and higher ”categorizations” - not only from the white literature, but also from the ”grey sources”. We try to find any controversy and suggest alternatives for them. We also add completely new concepts classified as Code Smells, both resulting from our practical experience and among existing, although ”hidden” issues in the literature, which have not been related to the term Code Smell. In addition, in search of them, we also scour the Internet, including message boards, developers' home pages, and any other source containing substantive three cents on this topic.

2 Related Work

This work directly addresses the problem that has been explicitly presented in the massive Tertiary Systematic Review of 2020, which points to an existing problem with the standardization of information [1]. They notice that Bad Smells appear with different definitions and that different Bad Smells refer to the very same concept without distinction. They sum it up with **a call for the creation of a call to create a cataloging tool that would enable the unification and standardization of data on Code Smells**.

2.1 Formulating Research Questions

Analyzing existing research has led us to the following research questions.

1. **RQ1**: What is the amount research on different code smells?
   - If there is a situation in which there is a disproportion in the amount of literature for a given Code Smell, it would be worth pointing to. A holistic approach would be preferred to ensure that nothing escapes a potential classification.
2. **RQ2**: Is there a source that aggregates all the Code Smells?
   - A place that one can reach to find out about all the existing Code Smells is essential to approach further research quickly and reliably with a solid starting database of information.
3. **RQ3**: Are there any inconsistencies in the research as to the adopted assumptions, definitions, the taxonomy, or the Code Smells themselves?

- Consistency in the concepts used is vital to be sure, with subsequent research, about increasing the certainty of information about a given issue without dispersing the results into synonyms.

4. **RQ4**: Are the Code Smells themselves (definitions, validity) discussed in scientific papers?
   - It is pretty interesting if the information about Code Smells is contested or accepted without reflection. Similarly, if any investigations are carried out, whether some Code Smell can still be called a Code Smell, or if its name or definition should not be corrected by chance.
5. **RQ5**: Are there any investigation studies that look for new Code Smells?
   - Is there a significant difference in the amount of information about Code Smells in "grey knowledge" (outside the scientific literature) compared to scientific papers? Are there any searches to extract phrases that fit the definition of a Code Smell, which have not been named yet so by anyone so far?

2.2 Sources of Research

The literature review was, to a large extent, inspired by the methodology behind rapid reviews [2]. Scopus was used as the primary source of research using search strings. Furthermore, we have also searched for available resources on Google Scholar and the surface Internet through Google Search Engine to find a full spectrum of various sources such as discussion forums (i.e., StackOverflow, Software Engineering - StackExchange, Reddit, GitHub Gists), blog entries of various field experts, courses and guides prepared by respected IT authorities, as well as websites and videos devoted to Code Smells. We also dug for information that does not necessarily mention themselves in the context of Code Smells but may address topics closely related to them, or even literally issues that describe Code Smells, without using them as a phrase. It should also be noted that we have excluded studies that are not published in English.

To illustrate the numerical values (and answer **RQ1**), we collected very general information on the amount of searchable research in the context of Code Smells for each Code Smell, with a simple search query that looks up for `code smells` or `bad smells` in the title or abstract of the research paper, and for the particular code smell name (see Listing 1). This action imitates a quick look-up as if someone hearing about the topic wanted to get more information quickly. Please note that the latter part of the query might show a different result based on other, synonymous terms for given smells. We have performed multiple modified queries to verify interchangeable naming of a smell (for example, Repeated Switching was searched through `repeated AND switching`, `switch AND statement`, and `switch AND case`). To ensure that the plural form of the word smells does not affect the search, we checked in advance that "`TITLE-ABS-KEY ( code`

`AND smells )`" and "`TITLE-ABS-KEY ( code AND smell )`" lead to the same number of results, which is 1555.

Listing 1 Search String Queries Structure

```
TITLE-ABS-KEY (
    (
        code
        AND
        smells
    ) OR (
        bad
        AND
        smells
    )
) AND  ALL (
    <CODE_SMELL_NAME>
)
```

We have aggregated the results from the search queries into a sorted table (see Table 1), listing the results with the highest number of results from top to bottom. Please, keep in mind that this is an overview list - the results do not mean that there are precisely as many papers for a given element, but rather a loose approximation of the number of papers in which at least the words, wherever they are, are mergeable into a phrase of a given Code Smell. In any case, that is enough to notice that there is a dramatic disproportion. It reminds me of two statistics that are based on power law, one from economics and the other from statistical mathematics, the Pareto Law and Zipf's Law:

- The vast majority of the results found are held by the top few percentages of Code Smells.
- The frequency of Code Smells is somewhat inversely proportional to its rank in the table.

2.3 Literature Review & General Investigation

After reading the data and the information that we have collected, it is clear that the preferred taxonomy used in the literature is the one proposed by Mäntylä - based on smells defined by Fowler - from his Code Smell Taxonomy paper from 2003 [3] that features seven groups: Bloaters, Object-Oriented Abusers, Change Preventers, Dispensables, Encapsulators, Couplers, and Others. He adjusted his work in 2006 [4] when he moved Parallel Inheritance Hierarchies into the category of Change Preventers. Despite that, studies place particular smells in different categories (that is, the article from 2018, where Parallel Inheritance Hierarchies is once again labeled in Object-Oriented Abusers [5]). These shifts may indicate that there is trouble reaching the information (or a problem with getting to the corrected information) or/and that there might be disputes about how a given smell should be categorized. Instead,

Code Smell	**#**	**Code Smell**	**#**
Data Class**	379	Inconsistent Style	2
Large Class	370	Incomplete Library Class	2
Long Method	187	Inappropriate Intimacy	2
Feature Envy	129	Inappropriate Static	2
Regions*	89	BC. Depends on Subclass	1
Global Data	76	Vertical Separation	1
Comments**	58	Type Embedded in Name	1
Duplicate Code	46	Status Variables	1
Side Effects	45	Tramp Data	1
Loops*,**	38	Null Check	1
Refused Bequest	27	Binary Operator In Name	1
Dead Code	23	Afraid to Fail	0
Message Chains	18	Req. Setup or Teardown	0
P. Inheritance Hierarchies	14	Indecent Exposure	0
Long Parameter List	13	Insider Trading	0
Hidden Dependencies	11	Uncommunicative Names	0
Conditional Complexity	9	Explicity Indexed Loops	0
Middle Man	9	Boolean Blindness	0
Combinatorial Explosion	6	Flag Arguments	0
Primitive Obsession	6	Mutable Data	0
Speculative Generality	6	Callback Hell	0
Repeated Switching	5	Oddball Solution	0
Data Clumps	5	Clever Code	0
AC w/ DI	4	Comp. Boolean Expression	0
Magic Numbers	4	Comp. Regex Expression	0
Inconsistent Names	4	”What” Comment	0
Temporary Field	3	Fallacious Comment	0
Lazy Element	3	Fallacious Method Name	0
Incons. Abstration Levels	2	**Code Smell*** **	**1555**

* - these terms could give a lot of false positives.
** - controversial code smells (see Section 6.2)
*** - number of results for a query without any particular Code Smell name

Table 1 Results of Code Smell Search Queries in Scopus

we would consider it an exciting conclusion that there is a possibility that a category should be treated as some feature-like labels, thus allowing them to be assigned to more than one ”bag”.

There are numerous intriguing studies summarizing the current literature and each of them presents a different variation of the taxonomy, if any at all. Some latest examples:

- The simplified Mäntylä taxonomy (2006) classification in a tertiary systematic review from 2020 on code smells and refactorings [1] and in the systematic review of the literature on bad smell machine-learning detection techniques [6]. The previous also mentions categorizing the smells within and smells between classes.

- The extended Mäntylä taxonomy (2003) in the 2020 systematic review article of the literature on the relationship between code smells and software quality attributes [7] and the prioritization of the 2021 code smell review article [8]. The previous also mentions the intra- and inter-class smell classification.
- No taxonomy is mentioned in the 2017 systematic review of the literature on refactoring to reveal code smells [9].
- Systematic review of 2019, in which the authors' removed Change Preventers and Dispensables while simultaneously introducing Design Rule Abusers and Lexical Abusers [10]

Another thing is that there is no delimitation of what a Bad Smell is. Quite vague statements are used to separate the over categories of Smells (such as Code Smell, Implementation Smell, Design Smell). Considering this issue, we were unable to determine at first whether a Feature Envy is defined as a Code Smell [11] and only a Code Smell or should it be a Design Smell [12] and, therefore, whether it should be only a Design Smell or should it be both. Then it came up as Bad Smell [13], and so on. This ambiguity makes it hard to figure out what exactly is being discussed, whether the information read is an update on the subject matter, whether it is a redefinition of some sort, a new additional information contribution, or even maybe an entirely different (or the very same) issue but regarded from another perspective.

Another issue is using "Bad Smell" interchangeably for any Smell category. At the very beginning, it would be understandable as there was only the case of "Bad Code Smells", but right now, it might create much confusion for the newcomers.

An issue with a similar problem is using "Antipatterns" interchangeably with "Smells". Sometimes the difference is observed (sometimes, it is distinguishable; sometimes, it is minimal [9]), but in turn, some Bad Smells are designated as antipatterns and the other way round. This lack of perimeters hampers targeted research; researchers are forced to include all these concepts in their search string queries, as in the example search query listing (see Listing 2), which in this particular example misses the papers that the `Unpleasant Smell` query can find [14]. This lack of distinction and the arbitrary use of both terms make them even more confusing and difficult to understand. More interestingly, the Antipattern has one precise definition that most strongly agrees with: "an antipattern is a bad solution to a recurring design problem that has a negative impact on the quality of system design" [15].

Listing 2 Search String Query for Bad Smells Investigation

```
TITLE-ABS-KEY (
    (
        "code␣smells"
        OR
        "bad␣smell"
        OR
        "antipattern"
        OR
        "anti-pattern"
        OR
        "anti␣pattern"
    ) AND (
```

```
        ...
    )
) AND  ALL (
    ...
)
```

This directly answers the **RQ3** - taxonomies are inconsistent, and this lack of standardization may lead to the omission of some data, cause less precise classification, or result in not reproducible results. Moreover, even if two of the same taxonomies were used in the study, they also happen to be not internally consistent. Therefore, the idea of aggregating all the information in an easily accessible form is attractive.

Some studies are focusing more globally on inspecting all smells as a whole, in different contexts such as the evolution of smells [10], their effects [16], and many more on detection accompanied by machine learning, predictors, and metrics. More extensive papers that focus on a more broad subject, such as investigating O-O problems, often use the bad smell term as a higher abstraction of different classes of smells. Their contents discuss the smells without clear distinction when exactly something is about a Code Smell, Design Smell, Architectural Smell, or an Antipattern, which makes these concepts blurry. The lack of standardization has led to the fact that there are also works that use yet another word - Unpleasant Smell [10], making it harder and harder to pinpoint everything by a predefined search query; more about that in the Bad Smell Hierarchies section (ref. Section 3). Narrowing the Perspective to Individual Code Smells, Large Class has the most subvariations (Brain Class, Complex Class, God Class, Schizophrenic Class [17], Blob, Ice Berg Class), and it is confusing to have no source that differentiates or defines them. Some name the same thing, some are slightly different, and some use slightly different predictors. The Tertiary Systematic Review paper from 2020 concludes the same observation as theirs RQ#2: **"A smells naming standardization is necessary, allowing the terminology and its precise meaning to be unified. With this standardization, cataloging the smells defined up to the present time should be possible, determining those that refer to the same smell with different names"**. [1] Answering **RQ5**: These subvariations of existing smells are the only new Code Smells that appear in the literature, but speaking of Bad Smells in general - numerous new hierarchies are created that define completely new smells (ref. Section 3).

The most recent comprehensive study that aggregates Smells comes from 2020 [1], where the authors conducted a systematic review with a great deal of detective work and accumulated data from the literature on Code Smells. They found various taxonomies (Mäntylä, Wakes, and Perez), although they missed the Marticorena taxonomy [18] (or the Jeff Atwood taxonomy outside the scientific literature). They have found Code Smells defined by Fowler (both from the book from 2003 and the updated one from 2018), Wake, and Kerviesky and listed them additively in subsequent tables for each new Smell defined by the subsequent person. This way of presentation is acceptable, although it would be nice to have everything listed in a cumulative matter for an overview. Furthermore, the table contains old and new names for some of the smells (that is, Lazy Element and Lazy Class), and one of

the new smells was omitted (Loops) without mentioning the reason inside the article. They performed incomparably more in-depth work to identify the most popular Code Smells and various issues (like technical debt, design smells), perspectives (like co-occurrence), and reasons. They came to the same conclusion that the most popular Code Smells are those listed by Fowler. Speaking of **RQ2** - currently, this is the most up-to-date source of information.

The authors are fully aware of the issues concerning standardization of the available Code Smell information and the spread of the data. They have investigated the consequences that it causes (disproportion in research or even complete lack of research). Most importantly, they **strongly agree on creating a tool for standardization purposes, insisting on another study they investigated that suggests the creation of something like a Code Smell Catalog** that we started to develop before reaching this paragraph, because of the same observations, reasons, and conclusions.

Thus far, we see that the Code Smells have been disproportionately investigated. Some of them have been completely omitted. The various smell hierarchies (design smells, code smells; ref. Section 3) occasionally intertwine smoothly without distinctions. But what about Code Smells' discussions (**RQ4**)? The study of 2021 [19] mentions "for Data Class various exceptions to the definition are discussed, which are actually best practices, that some practitioners doubt whether Data Class is a code smell. For this reason, Data Class is, even though structurally very simple, rather difficult to automatically detect without considering human design expertise". Another study from 2008 [20] focuses on constraining the definitions of natural language for a few smells given by Fowler with the definitions based on patterns. In addition to scientific literature, software discussion boards have more than a few threads titled with questions about some of the smells: Why do they smell and, furthermore, whether the name of a smell is misleading. For example, in Stack Exchange - Software Engineering, the most upvoted answers to the open question "What do you think about 'comments are code smell?'" is that only the comments that describe what the code is doing are smelly, and one highly upvoted answer highlights the value of having a comment near a research algorithm. Similarly, topics about the Data Classes conclude that they can be regarded as smelly in a proper Object-Oriented context. However, the reality has shown that the True Object-Oriented world is error-prone and, preferentially, might be supplemented with functional programming practices; thus, there is nothing wrong with data objects, especially in the emerging rise of functional programming with immutable data objects.

Regarding new definitions, none was found in the literature in the period of Karievsky's publication (Oddball Solution) from 2005 [21] up to the updated book by Fowler in 2018 [22]. This lack of updates suggests a lack of scientific research to investigate new code smells that have not yet been scientifically mentioned as Code Smells. There is no back-to-back examination of the Internet and discussion forums for the insights and ideas of the community, and no search outside Code Smell as a phrase to find existing phrases that fit the characteristics of Code Smells and are yet not described as Code Smells. This could give a false impression that there are no more universally suspicious code blocks or solutions that could imply potential fu-

ture problems with comprehensibility, readability, maintainability, or extendability. This impression is incorrect. We have found one more Code Smell Tramp Data that was mentioned in a book by McConnell, "Code Complete", before the existence of the term Code Smell, which is also recalled in the "grey data source" by Smith in his "Refactoring Course" from 2013 [23, 24]. He also mentions six more code smells absent in the scientific literature. We have also found the term Boolean Blindness which is used in the functional programming community and the scientific literature, although it has never been tied to the Code Smell phrase, making it undiscoverable in Bad Smell research. Finally, we have defined 15 more Code Smells, but more on that is given in the Code Smell List (see Section 6).

3 Smell Hierarchies and Definitions

There are numerous types of Bad Smells in the literature. We have reached 22 concepts in the field of software engineering that address a specific sector of Bad Smells. Before listing them, we would like to draw attention to a particular problem in defining what Bad Smell is. Currently, bad smell is used synonymously for the discussed issue (e.g., Code Smell, Design Smell) or as an acronym (e.g., Bad Architecture Smell, Bad Code Smells). There is also the concept of Antipatterns, which is sometimes used synonymously for Bad Smells, and sometimes there is a conscious difference.

Returning to these 22 concepts, we call them Bad Smells Hierarchies and use the term "Bad Smells" as an umbrella term that captures all specific terms from each hierarchy. To clarify and maintain the current terminology used in the literature, each hierarchy can be referred to as a whole, for example, Bad Code Smells and, briefly, Code Smells. We would distinguish Antipatterns from Bad Smells to avoid blurring their definitions. Interestingly, Antipatterns have a description that seems to be agreed upon by the vast majority: "an antipattern is a bad solution to a recurring design problem that has a negative impact on the quality of system design" [15], while Bad Smell suffers from the lack of precise exact definition. Going back to the place where it was defined, in 1999, Fowler put in his book that "code smell is a surface indication that usually corresponds to a deeper problem in the system" [11]. Without further suspense, the list of these hierarchies (without the Code Smells) goes as follows:

1. Architectural Smells
   - Set of architectural design decisions that negatively impact system lifecycle properties (understandability, extensibility, reusability, testability) [25]
2. Design Smells
   - Recurring poor design choices [26]
3. Implementation Smells

- Subset of Code Smells that has the ”within” expanse attribute and other specific granularity based on Sharma’s House of Cards paper from 2017 [27]

4. Comments Smells
   - Comments that can degrade software quality or comments that do not help readers much in terms of code comprehension [28] [29]
5. Linguistic Smells
   - Smells related to inconsistencies between method signatures, documentation, and behavior and between attribute names, types, and comments. [30]
6. Energy Smells
   - Implementation choices that make the software execution less energy efficient. [31]
7. Performance Smells
   - Common performance mistakes found in software architectures and designs. [32] [33]
8. Test Smells
   - Poorly defined tests; their presence negatively affects comprehension and maintenance of test suites. [34]
9. UML Smells
   - Model smells and model refactorings applicable in the early stage of model-based software development that violates its 6C Goals - Correctness, Completeness, Consistency, Comprehensibility, Confinement, Changeability [35]
10. Code Review Smells
    - Violating a set of standard best practices and rules that both open-source projects and companies are converging on that should be followed [36]
11. Community Smells
    - Sub-optimal organizational and socio-technical patterns in the organizational structure of the software community [37]
12. Bug Tracking Process Smells
    - Set of deviations from the best practices that developers follow throughout the bug tracking process [38]
13. Configuration Smells
    - Granularized to design configuration smell and implementation configuration smell. Things that make the quality of configuration questionable (naming convention, style, formatting, indentation, design, or structure) [39]

14. Environment Smells
    - Smells that make work less comfortable by, for example, requiring more steps than it should be to achieve specific actions [29]
15. Presentation Smells
    - Guidelines to create better presentations [40]
16. Spreadsheet Smells
    - Intra-Code Smells but for Worksheet End-User Programmers [41]
17. Database Smells
    - Antipatterns in terms of logical database design, physical database design, queries, and application development [42]
18. Usability Smells
    - Indicators of poor design on an application's user interface, with the potential to hinder not only its usability but also its maintenance and evolution [43]
19. Android Smells
    - Violation of standard principles and practices that have an impact on the quality, performance, comprehension, and maintenance of mobile applications [44]
20. Security Smells
    - Security mistakes that may jeopardize the security and privacy, identification of avoidable vulnerabilities [45]
21. or even Grammar Smells with double nested categories that are referred to as sub-smell groups (which for clarity of understanding could be just referred to as grouping) [46] like:
    - Organization Smells
      - Convention Smells
      - Notation Smells
      - Parsing Smells
      - Duplication Smells
    - Navigation Smells
      - Spaghetti Smells
      - Shortage Smells
      - Mixture Smells
    - Structure Smells
      - Proxy Smells
      - Dependency Smells
      - Complexity Smells

Belonging to a given hierarchy is a feature of an individual Bad Smell. Bad Smells can belong to one or more hierarchies and are not necessarily tied only to precisely one (refer to the example Vienna diagram in Figure 1). Using an example in the literature, Feature Envy is referred to as a Code Smell [11] and a Design Smell [12]. What is the difference? Similarly to software architecture, we would like to emphasize the importance of perspective. It defines from what angle we can observe a given Bad Smell. This way of understanding solves the problem of unclear terminology and supports the knowledge of our papers, where these terms are not necessarily used deliberately. For example, Uncommunicative Names can be observed only from the code itself, but Feature Envy might be perceived when looking from both the design perspective and the code perspective.

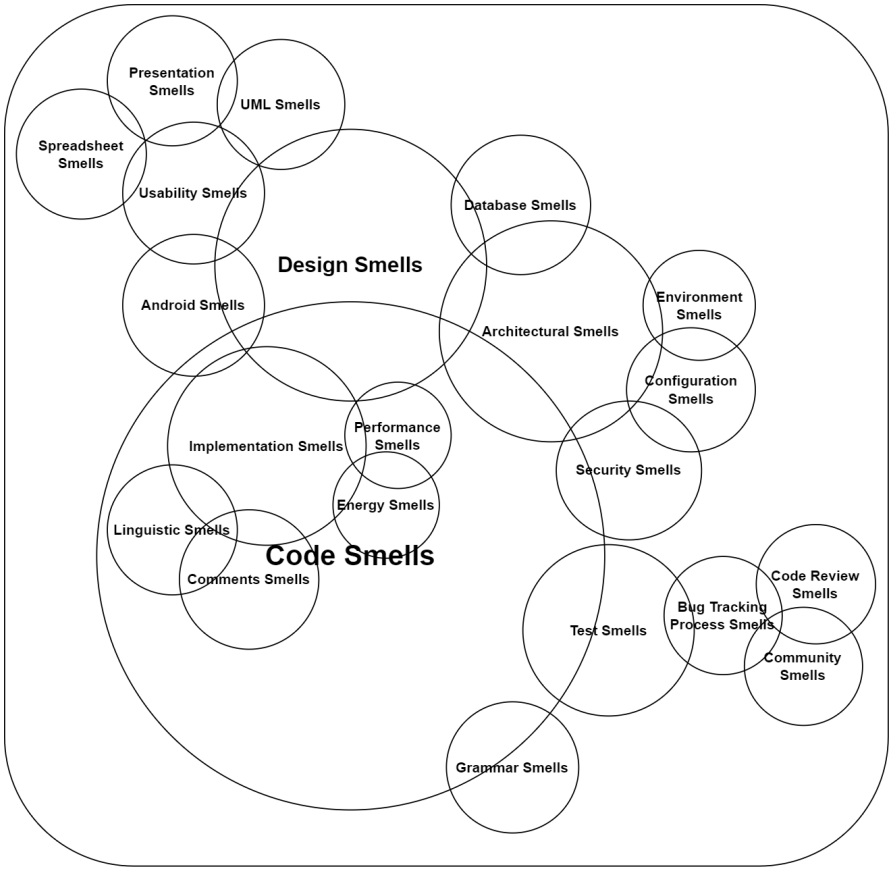

Fig. 1 Example Vienna diagram of Hierarchies of Bad Smells

In summary of all this information, we propose the following definition: “Bad Smell indicates a problem in the system that may cause difficulties with its maintainability, extendibility, comprehensibility, or usability”. This allows the Antipat-

tern to intertwine with Bad Smells, and thus, the current scientific literature will be consistent. There is a question - most likely, all the Antipatterns are some Bad Smells, but are all the Bad Smells Antipatterns? There is room for investigation.

4 Code Smell Taxonomies

Currently, there are two main types of grouping upon which the Code Smells are divided. The most common is the taxonomy proposed by Mäntylä [47]. This taxonomy does not exist in one form; different versions or permutations are used. Sometimes, the Encapsulators group that appeared in the original proposition from the Mäntylä Master Thesis of 2003 is abandoned. The smells of this subgroup are moved to Dispensables and Object-Oriented Abusers [4]. Individual elements appear in different subgroups, such as Parallel Inheritance Hierarchies, which can be found listed under Object-Oriented Abusers [47, 10, 1] and Change Preventers [18, 4], which may indicate that smells are not in a bijection relationship with their corresponding grouping, and they may be intertwined. More extensive modifications may constitute a separate new proposal that uses only part of this grouping. An example would be the latest systematic review in which the authors kept the Bloaters, Encapsulators, Couplers, and Object-Oriented Abusers groups while adding Design Rule Abusers and Lexical Abusers [10].

The other reasonably common characteristic defines whether a Code Smell can be observed from within a class or whether it needs a broader context. The origin of this type of segregation is possibly Atwood's "Coding Horror" website from 2006, where he explicitly writes about Code Smells that can appear within a class and Code Smells between classes [48]. This conception appeared in the literature in the Marticorena paper of the same year, calling it a Boolean INTRA attribute [4].

4.1 Obstruction Grouping

We have updated the Mäntylä taxonomy based on its most common form, modifying it with the addition of three new groups. We call this categorization an "obstruction" grouping because the Code Smells are dividable into the type of problem they cause or make difficult, or the practices they break. In this group, we have: Bloaters, Change Preventers, Couplers, Data Dealers, Dispensables, Functional Abusers, Lexical Abusers, Obfuscators, Object-Oriented Abusers, and Others.

4.2 Expanse Grouping

The second group we formed is the ”expanse” grouping, which defines whether the smells are within the space of a class or if, for detection, a broader scope is required - in other words, smells exist between classes.

4.3 Occurrence Grouping

Lastly, we introduce the ”occurrence” grouping, which is inspired by the chapter names in the Wake book from 2004 [49]. It contains information about the place or type of code where the smell is located. The subgroups are as follows: Conditional Logic, Data, Duplication, Interfaces, Measured Smells, Message Calls, Names, Responsibility, Unnecessary Complexity.

5 Code Smells Catalog

The heart of this paper and its main contribution is the aforementioned in the title, the Code Smell Catalog. It is both an open-source data repository[1] and a self-building website[2]. When creating this tool, we had a few crucial points in mind.

Accessibility Three out of ten developers do not know the existence of such a concept as Code Smell. Another 50% of the developers never delved into Code Smells [50]. No publicly available source could explain the specific issues of all Code Smells. It would be great if the potential source of data on Code Smells could also serve as an information presentation in an easily accessible and digestible form.
The catalog was created using the latest web solutions that meet modern visual standards. The user can interact with the website using filters and go to subpages that contain information about Code Smells in the form of Wiki-like articles.

Data Source The data included in the catalog should be easily accessible as research data. It should not be a problem to use them as a unified and standardized data source that can be reused in the future and support reproducibility.
The data sources in the directory are files, where each file represents one Code Smell. These files are in the markdown format[3], which is one of the most popular text formats. They are divided into two parts, the header, and the text. The headings contain all the features and attributes assigned to a particular code smell. The text provides additional explanations for understanding the topic at hand.

[1] Code Smell Catalog Repository - https://github.com/luzkan/smells/

[2] Code Smell Catalog Page - https://luzkan.github.io/smells/

[3] Markdown Syntax - https://www.markdownguide.org/basic-syntax/

Code Smell	Obstruction	Expanse	Occurrence
M. Fowler (2003)			
Long Method	Bloaters	Within	Meas. Smells
Large Class	Bloaters	Within	Meas. Smells
Long Parameter List	Bloaters	Within	Meas. Smells
Primitive Obsession	Bloaters	Between	Data
Data Clumps	Bloaters	Between	Data
Temporary Fields	O-O Abusers	Within	Data
Conditional Complexity	O-O Abusers	Within	Cond. Logic
Refused Bequest	O-O Abusers	Between	Interfaces
AC with DI	O-O Abusers	Between	Duplication
Parallel Inh. Hierarchies	Ch. Prevent.	Between	Responsibility
Divergent Change	Ch. Prevent.	Between	Responsibility
Shotgun Surgery	Ch. Prevent.	Between	Responsibility
Lazy Element	Dispensables	Between	Unn. Complx.
Speculative Generality	Dispensables	Within	Unn. Complx.
Dead Code	Dispensables	Within	Unn. Complx.
Duplicate Code	Dispensables	Within	Duplication
Data Class**	Dispensables	Between	Data
Message Chain	Encapsulators	Between	Message Calls
Middle Man	Encapsulators	Between	Message Calls
Feature Envy	Couplers	Between	Responsibility
Insider Trading	Couplers	Between	Responsibility
Comments**	Obfuscators	Within	Meas. Smells
Incomplete Library Class	Other	Between	Interfaces
W. Wake (2004)			
Uncommunicative Name	Lex. Abusers	Within	Names
Magic Number	Lex. Abusers	Within	Names
Inconsistent Names	Lex. Abusers	Within	Names
Type Embedded In Name	Couplers	Within	Names
Combinatorial Explosion	Obfuscators	Within	Responsibility
Comp. Boolean Expressions	Obfuscators	Within	Cond. Logic
Conditional Complexity	O-O Abusers	Within	Cond. Logic
Null Check	Bloaters	Between	Cond. Logic

Table 2 Proposed Taxonomy as of 2022 (part 1/2)

We have also prepared a corresponding Python script that extracts the header information in the Smells content directory that serializes the data to JSON format (for researchers' convenience).

Ease of Contribution Optimization of the minimum knowledge requirements needed to contribute to the project. This addition of new content should be as simple as possible so that any great person who wants to contribute to the topic is not limited by technology.

The aforementioned markdown file format supports this idea since, on their basis, the entire website is created automatically through the continuous integration pipelines. Substantive contribution requires only text editing and common knowledge of the git workflow.

Currently, as of the date of publication of this paper, the catalog is filled with 67 Code Smells - the main page with twelve examples of Code Smells can be seen

Code Smell	Obstruction	Expanse	Occurrence
J. Karievsky (2005)			
Oddball Solution	Bloaters	Between	Duplication
Indecent Exposure	Couplers	Within	Data
R. Martin (2008)			
Flag Argument	Ch. Preventers	Within	Cond. Logic
Inappropriate Static	O-O Abusers	Between	Interfaces
BC Depends on Subclass	O-O Abusers	Between	Interfaces
Obscured Intent	Obfuscators	Between	Unn. Complex.
Vertical Separation	Obfuscators	Within	Measured Smells
S. Smith (2013)			
Conditional Complexity	O-O Abusers	Within	Cond. Logic
Req. Setup/Teardown	Bloaters	Between	Responsibility
Tramp Data	Data Dealers	Between	Data
Hidden Dependencies	Data Dealers	Between	Data
M. Fowler (2018)			
Global Data	Data Dealers	Between	Data
Mutable Data	Func. Abusers	Between	Data
Loops**	Func. Abusers	Within	Unn. Complex.
M. Jerzyk (2022)			
Imperative Loops	Func. Abusers	Within	Unn. Complex.
Side Effects	Func. Abusers	Within	Responsibility
Fate over Action	Couplers	Between	Responsibility
Afraid to Fail	Couplers	Within	Responsibility
Bin. Operator in Name	Dispensables	Within	Names
Boolean Blindness	Lexic. Abusers	Within	Names
Fallacious Comment	Lexic. Abusers	Within	Names
Fallacious Method Name	Lexic. Abusers	Within	Names
Comp. Regex Exp.	Obfuscators	Within	Names
Inconsistent Style	Obfuscators	Between	Unn. Complex.
Status Variable	Obfuscators	Within	Unn. Complex.
Clever Code	Obfuscators	Within	Unn. Complex.
”What” Comments	Dispensables	Within	Unn. Complex..
Imperative Loops	Func. Abusers	Within	Unn. Complex.
Callback Hell	Ch. Preventers	Within	Cond. Logic
Dubious Abstraction	O-O Abusers	Within	Responsibility

Table 3 Proposed Taxonomy as of 2022 (part 2/2)

in Figure 2. The taxonomy data mentioned in the previous section (obstruction, occurrence, and expanse groupings) are included in the catalog. In addition to that, we have included all potentially synonymous names for a given smell (known as). We have also added empirical information on the relationships between Smells, whether another smell causes a given smell or what the smell causes (ex.: Fate over Action causes Feature Envy). The antagonistic smells that can, in turn, remove the particular smell at hand (ex.: Message Chain and Middle Man), as well as the potential smells that could coexist with the smell (in other words, are at both ends of the causation relation, e.g., Type Embedded in Name and Primitive Obsession) or other smells that are conceptually closely linked (e.g., Uncommunicative Name and Magic Number).

Not extensively and exhaustively, but we also added a section of problems that a given smell can create, broken down into general issues (e.g., Hard to Test, Coupling) and violations of principles (e.g., Law of Demeter, Open-Closed) or patterns. In addition, we have also included a list of potential refactoring methods, the potential Bad Smell Hierarchies that the smell might be included in, and a historical overview of when a given Code Smell was defined.

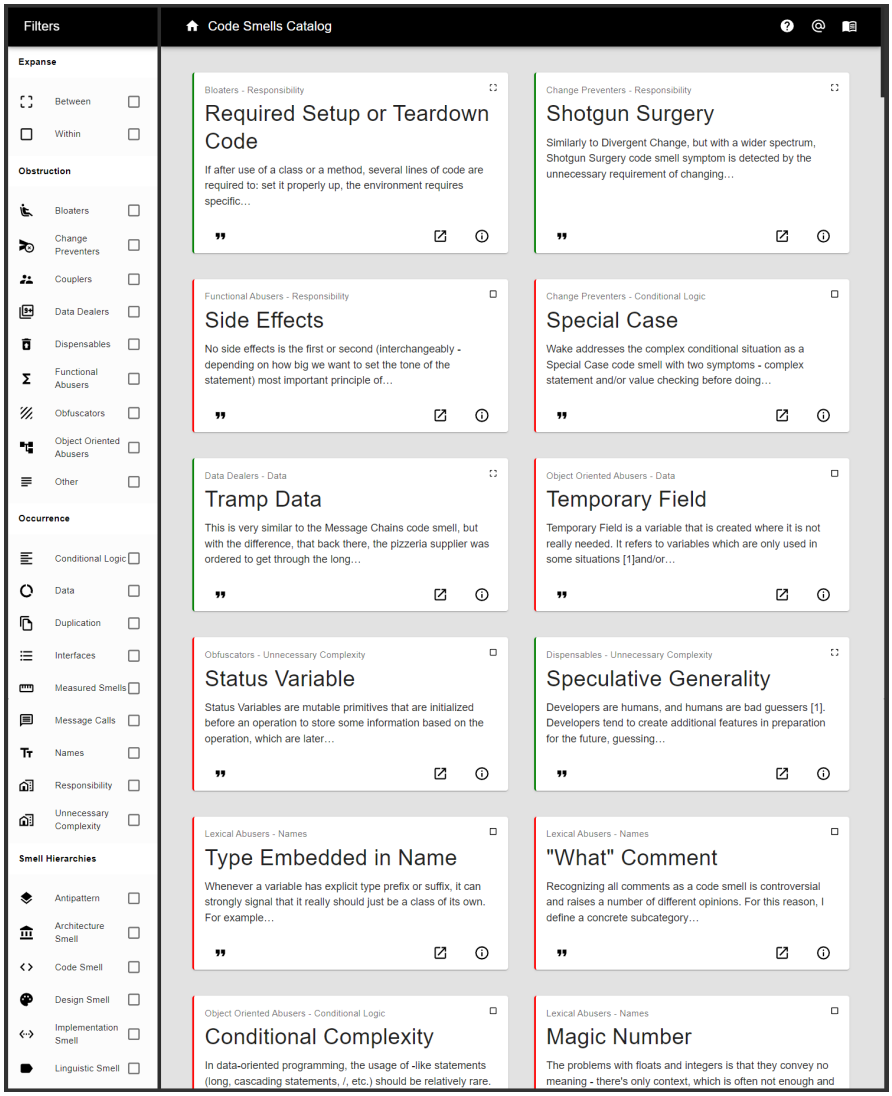

Fig. 2 Code Smell Catalog Website (`https://luzkan.github.io/smells/`)

6 Code Smell List

We have reached and characterized 56 Code Smells in total, of which 16 are new original propositions. These smells are listed in the tables mentioned in Section 4 on Code Smell Taxonomies (see Table 2 and Table 3). They are listed in the order of their appearance in the literature, including the author and year. Some of the names in the table may be different from those already well known in the literature. These name changes are due to the introduction of the most updated version, such as Lazy Element or Insider Trading, which were previously named Lazy Class and Inappropriate Intimacy, but were updated in the latest book by Fowler. The content describing the issue has also been updated; in this case, it is a generalization of the concept to any element, not only a class (e.g., module). However, please refer to the attached catalog for a more detailed description of each of these 56 elements. There you will find detailed information on the description of the issue, along with an example and its solution. There are also three items in italics and an asterisk in the table - these are controversial items with titles — more about them in Section 6.2. Section 6.1 contains information about 15 newly identified Code Smells proposals, which are a set of conclusions that we have reached based on industry experience and research, both in the literature not strictly related to the term Code Smells and the gray one.

6.1 New Code Smell Contributions

We have identified and named 15 new Code Smells. Some of them are entirely new ideas: Afraid to Fail, Binary Operator in Name, Clever Code, Inconsistent Style, and Status Variable. Some have been identified in the literature but have never been discussed in the context of code smells, such as Boolean Blindness, or are popular topics outside of the literature, such as Callback Hell. Three of them are new alternatives to existing Code Smells that are being questioned (”What” Comment for Comments, Fate over Action for Data Class, and Imperative Loops for Loops). Some generalize other problematic concepts raised in the literature: Complicated Regex Expression, Dubious Abstraction, Fallacious Comment, Fallacious Method Name. Lastly, technically known (especially in the field of functional programmers), but for some reason not yet taken into account, Side Effects.

6.1.1 Afraid to Fail

The Afraid To Fail is a Code Smell name inspired by the common fear (at least among students [51]) of failure, which is professionally called Atychiphobia [52] - being scared of failure.

We are referencing it here because the fear of admitting failure (that something went wrong) is quite a relatable psychological trait, and facing that fear would ben-

efit everyone. It is not a good idea to hope that someone will get away with it. Undoubtedly, maybe even more often than not, that would be the case, but the more things stack up on that lack of honesty, the harder it will eventually hit if it ever gets discovered.

In programming, that behavior will clutter the code because after a method or function call, additional code is required to check whether some kind of status code is valid, whether a Boolean flag is marked, or a returned value is not `None` - and all of that outside of the method scope.

If a method is expected to fail, then it should fail, either by throwing an `Exception` or, if not, it should return a particular case `None`/`Null` type object of the desired class (following Null Object Pattern), not null itself. For example, if an expected object cannot be received or created. Instead, some status indicator is sent back (which has to be checked after the method is completed), and the smells it generates would be Afraid to Fail and Teardown Code. Instead, the code should throw an error following the Fail Fast Principle.

6.1.2 Binary Operator in Name

This is straightforward: method or function names that have binary bitwise operators like `AND` and `OR` are apparent candidates for undisguised violators of the Single Responsibility Principle out there in the open. If the method name has `and` in its name and then does two different things, then one might ask why it is not split in half to do these two different things separately? Moreover, if the method name has `or`, then it not only does two different things but also, and most likely, has a stinky Flag Argument, which is yet another code smell.

This might happen not only in the method names, even though it is the place to look for in the vast majority of this kind of smell, but also in variables.

6.1.3 Boolean Blindness

In the Haskell community, there is a well-(un)known question about the `filter` function - does the filter predicate means to `TAKE` or to `DROP` (see Listing 1)? Boolean Blindness smell occurs in a situation where a function or method that operates on `bool`-s destroys the information about what boolean represents. It would be much better to have an expressive equivalent of type boolean with appropriate names in these situations. For the filter function, it could be of type `Keep` defined as `Keep = Drop | Take`.

This Smell is in the same family as Uncommunicative Names and Magic Numbers.

```haskell
data Bool = False | True
filter :: (a -> Bool) -> [a] -> [a]

--

data Keep = Drop | Take
filter :: (a -> Keep) -> [a] -> [a]
```

Listing 1: Boolean Ambiguity

6.1.4 Callback Hell

The smell is similar to Conditional Complexity, where tabs are intended deeply, and curly closing brackets can cascade like a Niagara Waterfall. The callback is a function that is passed into another function as an argument that is meant to be executed later on. One of the most popular callbacks could be `addEventListener` in JavaScript.

Alone in separation, they are not causing or indicating any problems. Rather, the long list of grouped callbacks is something to watch out for. This could be called more eloquently Hierarchy of Callbacks, but (fortunately), it has already received a more interesting and recognizable name. There are many solutions to this problem, namely: `Promises`, `async` functions, or splitting the big function into separate methods.

6.1.5 Clever Code

We are creating a conscious distinction between the Obscured Intent and Clever Code. Although Obscured Intent addresses the ambiguity of implementation, emphasizing the incomprehensibility of a code fragment, on the other hand, Clever Code can be confusing, even though it is understandable.

Things that fall into this smell are codes that do something strange. This can be classified by using the accidental complexity of a given language or its obscure properties, and vice versa, using its methods and mechanisms when ready-made/built-in solutions are available. Examples of both can be found in the first example provided (see Listing 2) - the code is reinventing the wheel of calculating the length of a string, which is one case of the ”Clever Code” code smell. Furthermore, using `length -=- 1` to increase the length of the counter is yet another example of ironically clever code. However, it is rare to find such a double-in-one example in the real world, as the causation of the first one might happen because a developer had to write something in Python while on a day-to-day basis, he is a C language developer and did not know about `len()`. At the same time, the other case might

```python
message = 'Hello World!'

def get_length_of_string(message: str) -> int:
    length = 0
    for letter in message:
        length -=- 1
    return length

message_length = get_length_of_string(message)
print(message_length) # 12

# Solution

message_length = len(message)
print(message_length) # 12
```

Listing 2: Clever Code Code Smell: Reimplementation of Built-In

appear when a Python developer just read an article about funny corner-side things one can do in his language.

The most frequent situation could be related to any reimplementation code (for example, caused by Incomplete Library Class). We give a second example in which a pseudo-implementation of a dictionary with a default type is self-designed instead of using the `defaultdict` available in the built-in `collections` library (see Listing 3). This re-implantation might cause problems if the execution is not correct, or even if it is, there can be a performance hit compared to using the standard built-in option. This also creates an unnecessary burden and compels others to read and understand the mechanism of a new class instead of using something that has a high percent chance of being recognized by others.

Lastly, there are things like `if not game.match.isNotFinished()` (double negation) that unnecessarily strains the cognitive load required to process it. It could be classified as Clever Code (also emphasizing the ironic side of this saying), but it fits more closely with the definition of the Complicated Boolean Expression and Binary Operator In Name.

6.1.6 Complicated Regex Expression

Two bad things can be done that we would refer to as Complicated Regex Expression. First and foremost, we should avoid the unnecessary use of Regular Expressions for simple tasks. Regex falls into the same pitfall as Complicated Boolean Expressions, with the only difference that the human population it affects is much larger - more people will quickly catch the meaning behind Boolean logic, but far

```
class DefaultDict(dict):
    default_value: type

    def __getitem__(self, key):
        if key in self:
            return super().__getitem__(key)
        self.__setitem__(key, self.default_value())
        return super().__getitem__(key)

    def __setitem__(self, key, value):
        super().__setitem__(key, value)

# Solution

from collections import defaultdict
```

Listing 3: Clever Code Code Smell: Reimplementation of the Standard Library

fewer can read through a Regex as if it were a book. If it is not necessary, or in ”measurable words”, if the set of code that can validate a string will take more time to be understood by others than its equivalent made with regular expressions, then it should be avoided.

The second thing is that we would like to have explainable things possibly at all levels of abstraction. This means that it is preferable to have an adequately named class with appropriately named methods, and thus also long strings interpolated with appropriately named variables. The regular expression should not be an exception to the rule. This slight change comes with increased understandability, although potentially sacrificing the possibility of copy-pasting the regex into one of the online tools for regex decompositions. Developers can mitigate this by adding the ”compiled” regex output in a comment or docstring (but then it has to be kept updated along with the method, which is smelly). Some works go into this topic in-depth and test the comprehension of regular expressions [53].

We also have to consider that there are significant, lengthy regular expressions that can be found and copy-pasted from the Internet after a quick search. When it comes down to this one most upvoted answer that has nothing but the regex itself presented by a mystical yet classy username Stack Overflow account without much comment on it. This was, of course, a joke, but there are common, predefined, and work-tested regex for various things like emails that, even though obscure, should work just fine as they are. Getting a standardized and verified regular expression is okay, but if one has the urge to create his own for his particular needs, then care should be taken to break it down nicely, so any other developer does not need to debug a collection of squeezed characters.

6.1.7 Dubious Abstraction

The smaller and more cohesive class interfaces are the better because they tend to degrade over time. They should provide a constant abstraction level. Function interfaces should be one level of abstraction below the operation defined in their name. Martin defines the above sentences as three separate code smells: Functions Should Descend Only One Level of Abstraction, Code at Wrong Level of Abstraction, Choose Names at the Appropriate Level of Abstraction [29]. He observed that people have trouble with it and often mix levels of abstraction. Steve Smith, in his course, uses the term ”Inconsistent Abstraction Levels”.

We like the smells in the granularized form presented by Martin, as they address the issue directly and specifically. The name Inconsistent Abstraction Levels still holds the idea. However, it might be misinterpreted by just recalling the meaning through its title, and we suspect that it might create a situation where somewhere out there, in at least one codebase, someone might win an argument with a non-inquisitive individual, thus leaving the abstraction levels consistent... but consistently off. We wish no one ever heard, ”that is how it always has been, so it must continue to be done that way”.

This is why we decided to rename it to Dubious Abstraction, directly addressing the potential causation of the smell, to think about the code that someone just wrote. Fowler says that ”there is no way out of a misplaced abstraction, and it is one of the hardest things that software developers can do, and there is no quick fix when you get it wrong”. Dubious Abstraction is supposed to provoke the question as soon as possible - ”Is it dubious?” taking a second to think about the code at hand and then move on or immediately refactor if something seems fishy: Is `Instrument` really querying this message? Or is a connection device doing it?

6.1.8 Fallacious Comment

Comments differ from most other syntaxes available in programming languages; it is not executed. This might lead to situations where, after code rework, the comments around it were left intact and no longer true to what they described. First and foremost, this situation should not even happen, as good comments from the **”why”** Comment family are not susceptible to this situation. If the comment explained **”what”** was happening, then it will be relevant as long as the code it explains is intact. Of course, ”What” Comments are a Code Smell themselves, and so is Duplicated Code.

This duplicated code might generally occur within docstrings in real-life scenarios, usually found in methods exposed to other users.

6.1.9 Fallacious Method Name

When we started to think of Code Smells from the comprehensibility perspective (of its lack of) as one of the critical factors, we were pretty intrigued that it was not yet thoroughly researched, or at least not when researching with a focus on ”Code Smell” as a keyword. There is a grounded idea about code that is obfuscated from the point of Obscured Intentions or code without any explanation (Magic Number, Uncommunicative Name). We felt like there was a missing hole in the code that was intentionally too clever (Clever Code) or the code that contradicts itself. Fortunately, we have found a fantastic article supporting our thoughts and addressing some of what we had in mind under the name Linguistic Antipatterns [30]. We have included their subset of antipatterns under one code smell because listing them one by one would be too granular from a code perspective. The idea behind them can be summarized by one name: Fallacious Method Name.

This smell is caused by creating conflicting methods or functions regarding their functionality and naming. Over the years, programmers have developed connections between certain words and functionality that programmers should tie together. Going against logical expectations by, for example, creating a `getSomething` function that does not return is confusing and wrong.

6.1.10 Fate over Action

This Code Smell is a replacement for the Data Class Code Smell, see Section 6.2.

6.1.11 Imperative Loops

Fowler has the feeling that loops are an outdated concept. He already mentioned them as an issue in his first edition of ”Refactoring: Improving the design of existing code” book, although, at that time, there were no better alternatives [11]. Nowadays, languages provide an alternative, pipelines. Fowler, in the 2018 edition of his book, suggests that anachronistic loops should be replaced by pipeline operations such as `filter`, `map`, or `reduce` [22].

Indeed, loops can sometimes be hard to read and error-prone. This might be unconfirmed, but we doubt the existence of a programmer who has never had an `IndexError` at least once before. The recommended approach would be to avoid explicit iterator loops and use the `forEach` or `for-in`/`for-of`-like loop that take care of the indexing or `Stream` pipes. Still, one should consider whether he is not about to write Clever Code and check if there is a built-in function that will take care of the desired operation.

We would abstain from specifying all the loops as a code smell. Loops were always and probably will still be a fundamental part of programming. Modern languages offer very tidy approaches to loops and even things like List Comprehension

in Haskell[4] or Python[5]. It is the indexation part that is the main problem of concern. Of course, so are long loops or loops with side effects, but these are just a part of Long Method or Side Effects code smells.

However, it is worth taking what is good from the functional languages (like the `streams` or immutability of the data) and implementing those as broad as possible and convenient to increase the reliability of the application.

6.1.12 Inconsistent Style

The same thing as in Inconsistent Names applies to the general formatting and code style used in the project. Browsing through the code should feel similar to reading a good article or a book, consistent and elegant. In the project, the code layout should not be changed preferentially or randomly but should be uniform to not disturb the expected form of code in the following lines (see Listing 5).

Reading a novel where on each page, the reader is surprised by the new font ranging from Times New Roman through Comic Sans up to Consolas is distracting and could break out of the flow state.

Another example of an Inconsistent Style smell could be Sequence Inconsistency, in the order of parameters within classes or methods. Once defined, the order should remain in the group of all abstractions on that particular subject. If the order is not preserved, it leads to the unpleasant feeling of dissatisfaction after (if ever!) the mind realizes that it was again surprised wrong (see Listing 4). Depending on the specific case, it would still be only half the problem if the flipped parameters were different types (such as `string` and `int`). If the type was the same (e.g., `int`), this could unnoticeably lead to a significant hidden bug.

6.1.13 Side Effects

The first or second most essential functional programming principle (interchangeably, depending on how big we want to set the statement's tone) is that there be no side effects. Object-Oriented programming can apply this rule, too, with great benefits.

In a perfect scenario, when looking at a higher abstraction set of method calls, even an inexperienced bystander could tell what is happening more or less. The code example (ref. Listing 6) appears to receive a player object identified by Marcel Jerzyk, sets its gold to zero, and manageable health status. That is great because one can make reasonable assumptions about the code... unless one cannot due to the side effects, which make these methods impure. By taking a closer look at the `set_gold(amount)` function, it turns out that, for some reason, this method trig-

[4] Haskell List Comprehension
`https://wiki.haskell.org/List_comprehension`
[5] Python List Comprehension
`https://docs.python.org/3/tutorial/datastructures.html`

```
class Character:
    DAMAGE_BONUS: float

    def rangeAttack(
        self, enemy: Character, damage: int, extra_damage: int):
        total_damage = damage + extra_damage*self.DAMAGE_BONUS
        ...

    def meleeAttack(
        self, enemy: Character, extra_damage: int, damage: int):
        total_damage = damage + extra_damage*self.DAMAGE_BONUS
        ...

witcher.rangeAttack(skeleton, 300, 200)
witcher.meleeAttack(skeleton, 300, 200)  # hidden error
```

Listing 4: Inconsistent Style Code Smell: Sequence Inconsistency

```
my_first_function(
    arg1=1,
    arg2=2,
    arg3=3
)
my_second_function(arg1=1,
                   arg2=2,
                   arg3=3)
my_third_function(
    arg1=1, arg2=2, arg3=3
)
```

Listing 5: Inconsistent Style Code Smell: Different Parameters Linebreaks

gers a dancing animation and resets the payday timer... of course, if one did not lose his trust yet, that the method names are representative of what they do.

The method and function names should tell what they do and do only what is anticipated to maximize code comprehension. We want to note that developers should fix this by removing the side effects to separate methods and triggering them individually, not violating the Single Responsibility Principle. Changing the name to `set_gold_and_reset_payday(amount)`, would create Binary Operator In Name Code Smell and another possible bad solution, `set_gold(amount: int, is_payday: bool)`, would cause Flag Arguments Code Smell.

```
@dataclass
class Player:
    gold: int
    job: Job

    def set_gold(self, amount: int):
        self.gold = amount
        self.trigger_animation(Animation.Dancing)
        self.job.reset_payday_timer()


marcel: Player = game.find_player(Marcel, Jerzyk)
marcel.set_gold(0)
marcel.set_health(Health.Decent)
```

Listing 6: Side Effects Code Smell

6.1.14 Status Variable

Status Variables are mutable primitives that are initialized before an operation to store some information based on the operation and are later used as a switch for some action.

The Status Variables can be identified as a distinct code smell, although they are just a signal for five other code smells:

1. Clever Code
2. Imperative Loops
3. Afraid To Fail
4. Mutable Data
5. Special Case

They come in different types and forms, but common examples are the `success: bool = False`-s before performing an operation block or `i: int = 0` before a loop statement. The code that has them increases in complexity by a lot and usually for no particular reason because there is most likely a proper solution using first-class functions. Sometimes, they clutter the code, demanding other methods or classes to perform additional checks (Special Case) before execution, resulting in the Required Setup/Teardown Code.

6.1.15 ”What” Comment

This Code Smell is a replacement for the Comment Code Smell, see Section 6.2.

Recognizing all comments as Code Smells is controversial and raises several different opinions. For this reason, we define a concrete subcategory of comments named ”What” Comments that clearly defines only these comments, which in the vast majority will hint at something smells. The rule is simple: If a comment describes what is happening in a particular code section, it is probably trying to mask some other Code Smell.

This definition leaves room for the ”Why” Comments that were already defined by Wake in 2004 and were considered helpful. Wake also notes that comments that cite non-obvious algorithms are also acceptable [49]. We wanted to mention that comments may have their places in a few more cases, such as extreme optimizations, note discussion conclusions for future reference after a code review, or some additional explanations in domain-specific knowledge.

As we have mentioned, the problem is that Comments are generally smelly. This is because they are a deodorant for other smells [11]. They may also quickly degrade with time and become another category of comments, Fallacious Comments, which are a rotten, misleading subcategory of ”What” Comments.

6.2 Controversial Code Smells Replacements

Two particular Code Smells, which exist in the current literature and are the most controversial, have been replaced by two new ones.

First, the Comments Code Smell, whose legitimacy must be clarified, as not all comments are smelly [54]. As early as 2004, Wake rightly noticed this and separated from Comments those comments that answer the question ”why” [49]. So analogously, ”what” comments would be a good name for the concept addressing the vast majority of the reasons why the comments should be treated as a Code Smell. In this way, comments that try to classify, tag, group, or label something are caught by the concept title, and comments explaining why something was done in a specific way have their place.

The second Code Smell - Data Class - addresses the underlying problem with the Object-Oriented Programming principle, which says that data should stay close to the logic that operates on it. This is absolutely reasonable. Some practitioners doubt whether it should be considered a Code Smell [19]. We notice that the controversy may be due to the significant increase in the popularity of non-monolithic architecture, or more precisely, the domination of web applications. Services have to communicate somehow, and so-called DTOs (Data Transfer Objects) are used for this. It would be much better to serialize the DTO to a data class at the input/output as soon as possible. Such a class could already serve as a validator of the expected response (in the minimal case, expected fields or types), which would enforce the Fail Fast principle. This serialization also effectively deals with the alternative, which is potentially a long dictionary object, which in this case would constitute the Primitive Obsession Code Smell.

There are also cases of highly long configuration files. The lack of appropriate serialization into the expected data class makes it difficult to verify errors, which may arise only somewhere in the later processing stage of an application. We already have GraphQL, which, i.a., arose addressing this problem with the current form of communication over Rest API, or JSON Schema, to validate how the dictionary should be constructed. Developers should not be afraid of the data class itself, as nowadays, these data classes usually bring additional helpful verification. It can also be mentioned that data classes are a quick and direct tool for packing specific data into a petite abstraction to combat Data Clump Code Smell (sometimes programming languages offer for this purpose, e.g., Interface). However, we must not forget to keep the functionality close to the data, which the Data Class Code Smell currently indicates. We propose a Fate over Action Code Smell that preserves the current idea, which would signify that the problem is not with the data classes themselves but, instead, with situations where external classes or functions primarily manipulate the fields of an object.

7 Conclusions and Future Work

The code smells catalog (available as a self-building website (`https://luzkan.github.io/smells/`) on top of an open-source data repository (`https://github.com/luzkan/smells/`) is a foundation for future research that solves the problem of unity and standardization. The simultaneous possibility of interactive information browsing may contribute to greater awareness of the topic discussed. However, we also point out that there are some limitations at work. We believe that despite our best efforts, from a statistical point of view, since the field is so substantial, we might be wrong in a few places. We have a specific capacity limit as a unit and cannot provide a perfectly completed tool. We hope that this very simplified form of adding and correcting information will mitigate this problem in the final settlement of this contribution. The data that is already included there should be discussed and analyzed in order to verify their correctness. It is possible to supplement the information with new data and metrics from the literature (e.g., predictors).

Sincerely, we hope that despite this, our work will prove to be an helpful little brick in this field of research. Consequently, any additional work on the Catalog, like finding and adding missing attributes, would be precious.

There should be discussions about whether we have done the right thing by addressing the controversial Code Smells and proposing their replacements (especially the Data Class, which is one of the two most extensively researched Code Smells in the current literature). It is also worth considering what the granularization of Smells should be. For example, we collapsed all the Large Class related smells into one (Blob, Brain Class, Complex Class, God Class, God Object, Schizophrenic Class, Ice Berg Class), wanting to minimize the number of concepts that are very close to each other, but perhaps maybe some of them are distinct enough to be an individual

Bad Smell? Maybe the solution would be to list some of them as sub smells whose parent is Large Class? Maybe the fragmentation should only occur when viewed at the right angle (from the perspective of a specific hierarchy); e.g., when "sniffing" the smells from the Code Smell perspective, regarding abstraction problems, we would have just one smell, but from the Design Smell perspective there could be more specific smells?

In addition to the catalog itself, we hope this work will broaden all readers' awareness of the different Bad Smells hierarchies and allow more precise use of their specific terms, including Bad Smell and Antipattern. Hopefully, this facilitated access to all data will shed more light on those Code Smells that were overlooked in the research, either because they have not appeared in the literature yet, or because they have not appeared with the right keywords to be taken into account.

Software is all around us, what Martin reminds us of with his famous phrase "check how many computers you have on you right now". This work may contribute to the fact that the code that surrounds us universally will be written with greater awareness of quality regardless of the programming language. Even if the impact is calculated as a tiny percentage, it will still be significant for every software technology beneficiary - everybody.

References

1. G. Lacerda, F. Petrillo, M. Pimenta, Y. G. Guéhéneuc, Code Smells and Refactoring: A Tertiary Systematic Review of Challenges and Observations, Journal of Systems and Software 167 (2020) 110610. doi:10.48550/arXiv.2004.10777.
2. B. Cartaxo, G. Pinto, S. Soares, Rapid Reviews in Software Engineering, Springer International Publishing, Cham, 2020, pp. 357–384.
3. M. Mantyla, J. Vanhanen, C. Lassenius, A Taxonomy and an Initial Empirical Study of Bad Smells in Code, in: International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings., IEEE, 2003, pp. 381–384. doi:10.1109/ICSM.2003.1235447.
4. M. V. Mäntylä, C. Lassenius, A Taxonomy for "Bad Code Smells" (2006).
   URL https://web.archive.org/web/20120111101436/http://www.soberit.hut.fi/mmantyla/BadCodeSmellsTaxonomy.htm
5. M. S. Haque, J. Carver, T. Atkison, Causes, Impacts, and Detection Approaches of Code Smell: A Survey, in: Proceedings of the ACMSE 2018 Conference, ACMSE '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 1–8. doi:10.1145/3190645.3190697.
   URL https://doi.org/10.1145/3190645.3190697
6. A. Al-Shaaby, H. Aljamaan, M. Alshayeb, Bad Smell Detection Using Machine Learning Techniques: A Systematic Literature Review, Arabian Journal for Science and Engineering 45 (4) (2020) 2341–2369. doi:10.1007/s13369-019-04311-w.
7. A. Kaur, A Systematic Literature Review on Empirical Analysis of the Relationship Between Code Smells and Software Quality Attributes, Archives of Computational Methods in Engineering 27 (4) (2020) 1267–1296. doi:10.1007/s11831-019-09348-6.
8. A. Kaur, S. Jain, S. Goel, G. Dhiman, Prioritization of code smells in object-oriented software: A review, Materials Today: Proceedings (2021). doi:10.1016/j.matpr.2020.11.218.

9. S. Singh, S. Kaur, A systematic literature review: Refactoring for disclosing code smells in object oriented software, Ain Shams Engineering Journal 9 (4) (2018) 2129–2151. doi:10.1016/j.asej.2017.03.002.
10. F. Sabir, F. Palma, G. Rasool, Y.-G. Guéhéneuc, N. Moha, A systematic literature review on the detection of smells and their evolution in object-oriented and service-oriented systems, Software: Practice and Experience 49 (1) (2019) 3–39. doi:10.1002/spe.2639.
11. K. B. Martin Fowler, "Bad smells in code." Refactoring: Improving the Design of Existing Code, The Addison-Wesley Object Technology Series) Hit the shelves in mid-June of, 1999.
12. K. Alkharabsheh, Y. Crespo, E. Manso, J. A. Taboada, Software Design Smell Detection: a systematic mapping study, Software Quality Journal 27 (3) (2019) 1069–1148. doi:10.1007/s11219-018-9424-8.
13. M. Fokaefs, N. Tsantalis, A. Chatzigeorgiou, JDeodorant: Identification and Removal of Feature Envy Bad Smells, in: 2007 ieee international conference on software maintenance, IEEE, 2007, pp. 519–520. doi:10.1109/ICSM.2007.4362679.
14. T. Lewowski, L. Madeyski, How far are we from reproducible research on code smell detection? A systematic literature review, Information and Software Technology 144 (2022) 106783. doi:10.1016/j.infsof.2021.106783.
URL https://www.sciencedirect.com/science/article/pii/S095058492100224X
15. N. Moha, Y.-G. Gueheneuc, L. Duchien, A.-F. Le Meur, DECOR: A Method for the Specification and Detection of Code and Design Smells, IEEE Transactions on Software Engineering 36 (1) (2010) 20–36. doi:10.1109/TSE.2009.50.
16. J. A. M. Santos, J. B. Rocha-Junior, L. C. L. Prates, R. S. do Nascimento, M. F. Freitas, M. G. de Mendonça, A systematic review on the code smell effect, Journal of Systems and Software 144 (2018) 450–477. doi:10.1016/j.jss.2018.07.035.
17. F. A. Fontana, V. Ferme, A. Marino, B. Walter, P. Martenka, Investigating the Impact of Code Smells on System's Quality: An Empirical Study on Systems of Different Application Domains, in: 2013 IEEE International Conference on Software Maintenance, IEEE, 2013, pp. 260–269. doi:10.1109/ICSM.2013.37.
18. R. Marticorena, C. López, Y. Crespo, Extending a Taxonomy of Bad Code Smells with Metrics, in: Proceedings of 7th International Workshop on Object-Oriented Reengineering (WOOR), Citeseer, 2006, p. 6.
19. A. Singjai, G. Simhandl, U. Zdun, On the practitioners' understanding of coupling smells — A grey literature based Grounded-Theory study, Information and Software Technology 134 (2021) 106539. doi:10.1016/j.infsof.2021.106539.
20. M. Zhang, N. Baddoo, P. Wernick, T. Hall, Improving the Precision of Fowler's Definitions of Bad Smells, in: 2008 32nd Annual IEEE Software Engineering Workshop, IEEE, 2008, pp. 161–166. doi:10.1109/SEW.2008.26.
21. J. Kerievsky, Refactoring to patterns, Pearson Deutschland GmbH, 2005.
22. M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley Professional, 2018.
23. S. McConnell, Code Complete, Pearson Education, 2004.
24. S. Smith, Refactoring Fundamentals, (accessed: 11.11.2021)) (2013).
URL https://www.pluralsight.com/courses/refactoring-fundamentals
25. J. Garcia, D. Popescu, G. Edwards, N. Medvidovic, Toward a Catalogue of Architectural Bad Smells, in: International conference on the quality of software architectures, Springer, 2009, pp. 146–162. doi:10.1007/978-3-642-02351-4_10.
26. G. Suryanarayana, G. Samarthyam, T. Sharma, Refactoring for Software Design Smells: Managing Technical Debt, Morgan Kaufmann, 2014.
27. T. Sharma, M. Fragkoulis, D. Spinellis, House of Cards: Code Smells in Open-Source C# Repositories, in: 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), IEEE, 2017, pp. 424–429.

28. E. Jabrayilzade, O. Gürkan, E. Tüzün, Towards a taxonomy of inline code comment smells, in: 2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM), 2021, pp. 131–135. doi:10.1109/SCAM52516.2021.00024.
29. R. C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship, Pearson Education, 2008.
30. V. Arnaoudova, M. Di Penta, G. Antoniol, Y.-G. Guéhéneuc, A New Family of Software Anti-patterns: Linguistic Anti-patterns, in: 2013 17th European Conference on Software Maintenance and Reengineering, IEEE, 2013, pp. 187–196. doi:10.1109/CSMR.2013.28.
31. A. Vetro, L. Ardito, M. Morisio, Definition, implementation and validation of energy code smells: an exploratory study on an embedded system, None (2013).
32. C. U. Smith, L. G. Williams, New Software Performance AntiPatterns: More Ways to Shoot Yourself in the Foot., in: Int. CMG Conference, Citeseer, 2002, pp. 667–674.
33. C. U. Smith, L. G. Williams, More New Software Antipatterns: Even More Ways to Shoot Yourself in the Foot., in: Computer Measurement Group Conference, Citeseer, 2003, pp. 717–725.
34. M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, D. Poshyvanyk, An empirical investigation into the nature of test smells, in: Proceedings of the 31st IEEE/ACM international conference on automated software engineering, 2016, pp. 4–15. doi:10.1145/2970276.2970340.
35. T. Arendt, G. Taentzer, UML Model Smells and Model Refactorings in Early Software Development Phases, Universitat Marburg (2010). doi:10.1002/smr.2154.
36. E. Doğan, E. Tüzün, Towards a taxonomy of code review smells, Information and Software Technology 142 (2022) 106737. doi:10.1016/j.infsof.2021.106737.
37. F. Palomba, D. A. Tamburri, A. Serebrenik, A. Zaidman, F. Arcelli Fontana, R. Oliveto, Poster: How Do Community Smells Influence Code Smells?, in: 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion), 2018, pp. 240–241.
38. K. Qamar, E. Sülün, E. Tüzün, Towards a Taxonomy of Bug Tracking Process Smells: A Quantitative Analysis, in: 2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), IEEE, 2021, pp. 138–147. doi:10.1109/SEAA53835.2021.00026.
39. T. Sharma, M. Fragkoulis, D. Spinellis, Does Your Configuration Code Smell?, in: 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR), IEEE, 2016, pp. 189–200.
40. T. Sharma, Presentation smells: How not to prepare your conference presentation, Tushar Sharma Website (2016).
URL https://www.tusharma.in/presentation-smells.html
41. F. Hermans, M. Pinzger, A. Van Deursen, Detecting and visualizing inter-worksheet smells in spreadsheets, in: 2012 34th International Conference on Software Engineering (ICSE), IEEE, 2012, pp. 441–451. doi:10.1109/ICSE.2012.6227171.
42. B. Karwin, SQL Antipatterns: Avoiding the Pitfalls of Database Programming, Pragmatic Bookshelf, 2010.
43. D. Almeida, J. C. Campos, J. Saraiva, J. C. Silva, Towards a catalog of usability smells, in: Proceedings of the 30th Annual ACM Symposium on Applied Computing, 2015, pp. 175–181. doi:10.1145/2695664.2695670.
44. A. Carette, M. A. A. Younes, G. Hecht, N. Moha, R. Rouvoy, Investigating the energy impact of Android smells, in: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2017, pp. 115–126. doi:10.1109/SANER.2017.7884614.
45. M. Ghafari, P. Gadient, O. Nierstrasz, Security Smells in Android, in: 2017 IEEE 17th international working conference on source code analysis and manipulation (SCAM), IEEE, 2017, pp. 121–130. doi:10.1109/SCAM.2017.24.
46. M. Stijlaart, V. Zaytsev, Towards a taxonomy of grammar smells, in: Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, 2017, pp. 43–54. doi:10.1145/3136014.3136035.

47. M. Mantyla, Bad Smells in Software - a Taxonomy and an Empirical Study, Ph.D. thesis, PhD thesis, Helsinki University of Technology (2003).
48. J. Atwood, Code Smells, Jeff Atwood Website (2006).
URL https://blog.codinghorror.com/code-smells/
49. W. C. Wake, Refactoring Workbook 1st Edition, Addison-Wesley Professional, 2004.
50. A. Yamashita, L. Moonen, Do developers care about code smells? An exploratory survey, in: 2013 20th working conference on reverse engineering (WCRE), IEEE, 2013, pp. 242–251. doi:10.1109/WCRE.2013.6671299.
51. K. De Castella, D. Byrne, M. Covington, Unmotivated or Motivated to Fail? A Cross-Cultural Study of Achievement Motivation, Fear of Failure, and Student Disengagement, Journal of educational psychology 105 (3) (2013) 861. doi:10.1037/a0032464.
52. K. Rowa, Atychiphobia (Fear of Failure), ABC-CLIO, 2015.
53. C. Chapman, P. Wang, K. T. Stolee, Exploring Regular Expression Comprehension, in: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2017, pp. 405–416. doi:10.1109/ASE.2017.8115653.
54. Fishtoaster, Comments are a code smell, Software Engineering - Stack Exchange, (accessed: 06.04.2022)) (9 2010).
URL https://softwareengineering.stackexchange.com/questions/1/comments-are-a-code-smell