Wrocław University of Science and Technology
Faculty of Computer Science and Management

Field of study:	**Applied Computer Science**
Speciality:	**Software Engineering**

MASTER THESIS

Code Smells: A Comprehensive Online Catalog and Taxonomy

Marcel Jerzyk

Supervisor
dr hab. inż. Lech Madeyski, prof. uczelni

Keywords: Code Smells, Bad Smells, Taxonomies, Literature and Grey Data Summary, Catalog

WROCŁAW 2022

CONTENTS

ABSTRACT

CODE SMELLS: A COMPREHENSIVE ONLINE CATALOG AND TAXONOMY

Context: More than once, during intimate private conversations, curiosities from work were exchanged, and it is no different in the field of software engineers - "if you only would know what the code is like in my company". We all would like to trust the entire spectrum of devices that surround us today. There is one intuitive concept, an understanding of which allows one to create such a code. Code that one does not need to be ashamed of and which will make everyone's life better - the "Code Smells". This concept is also an important factor when measuring code quality and, by the way, technical debt. Their number in the code is directly correlating with the efficiency with which the program can be further developed or maintained. At the same time, although surprisingly, this concept is not widely known or understood among programmers. Somewhat indifferently in the scientific literature, it is scattered and non-standardized. New "adjacent" branches are being built upon this concept, but there is no place where they are aggregated and summarized. The mere identification of elements that can be characterized as "Code Smell" has ceased. The existing ones have a significant disproportion in the "headlight" obtained in the research area.

Objective: Goal (#1): The goal of this work is to create and provide a shared and widely accessible open-source *Code Smell Catalog* that enforces the ease of contribution so that minimal technical requirements are needed from contributors to provide new substantive information. This catalog can perform functions both useful for researchers as a unified, uniform data system, enabling immediate information extraction, and also for programmers with a wide range of experience: for juniors, as a source of knowledge, and for more experienced ones, as a set of practices, which can be sent back during discussions or code review disputes. *Goal (#2):* In addition, the aim is to identify all possible concepts that can be characterized as Code Smells and to identify and address any existing "Code Smells" controversies. *Goal (#3):* Finally - to characterize the Code Smells by assigning them appropriate features resulting from the literature's main, already existing taxonomies.

Method: I have researched both the literature with the already existing "code smell" connection (that is, Code Smells searchable, using queries related to `Code Smell`), but also the literature that talks about a concept that fits the Code Smell definition, but at the same time has never been mentioned along with it. I have also searched in the area of "gray data", including discussion forums, technical articles, developer communities, as well as blogs, and developers' websites. The results were analyzed and interpreted using the knowledge I gathered, classifying them and verifying their internal consistency.

Results: I identified 56 Code Smells, of which 15 are new and original propositions, based on the literature, gray data, and personal industry experience. Two existing Code Smells concepts, which aroused the greatest and justified controversies, were modified and adapted to the present state of knowledge. I have created an online catalog with successive results that meet its assumptions. The catalog has additional interactive filtering functionality based on taxonomies. The catalog also contains individual articles for each of the identified Code Smells. Each Code Smell was classified according to the taxonomy, synonyms (or other closely related smells), hierarchy, refactoring methods, the type of problems it causes (including: what programming principles it breaks), relation to other Code Smells, the hierarchy of Bad Smell it may be revealed in, a historical note, and original minimal "smelly" examples and their potential solution. Also, I have found and listed 22 different types of Bad Smells, which I call "hierarchies". I also drew attention to the vague distinction between the Bad Smell concepts and Antipatterns.

Conclusion: The Code Smells are a precious concept and should be important to everyone in the industry: developers (efficiency), product owners (quality), analysts (technical debt, pace of implementation) and clients (support and development), but it does not have a well-beaten path in the general consciousness. This work has the potential to raise awareness within the industry and correct gaps in the existing scientific literature. It will allow further research to be undertaken consciously because access to the accumulated information resource (thanks to the catalog) 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.

STRESZCZENIE

ZAPACHY KODU: KOMPLEKSOWY KATALOG ONLINE ORAZ TAKSONOMIA

Kontekst: Niejednokrotnie ludzie podczas bliskich, prywatnych rozmów wymieniają między sobą ciekawostki z pracy i nie inaczej jest w dziedzinie software engineering – *„gdybyście tylko wiedzieli, jaki wygląda kod w mojej firmie”.* Wszyscy chcielibyśmy móc zaufać całemu spektrum urządzeń, które w dzisiejszych czasach wszechobecnie nas otacza. Jednym takim intuicyjnym konceptem (choć dzięki badaniom coraz bardziej mierzalnym), którego zrozumienie pozwala na tworzenie takiego kodu, kodu którego nie trzeba się wstydzić, jest "code smell". Jest to ważny czynnik przy pomiarze jakości kodu, ale także chociażby długu technicznego. Ich liczba w kodzie bezpośrednio koreluje z tym, jak efektywnie program może być dalej rozwijany lub utrzymywany. Jednocześnie, choć o dziwo, pojęcie to nie jest powszechnie znane, bądź rozumiane wśród programistów. Nie lepiej jest również od strony literatury naukowej – zagadnienie jest rozproszone i nieustandaryzowane. Na tym koncepcie tworzone są nowe sąsiednie rozgałęzienia, ale nie ma miejsca, w których byłyby one zagregowane i ustandaryzowane. Sam ruch identyfikacji nowych elementów, które można scharakteryzować jako „brzydki zapach kodu” zanikł, a z kolei wśród tych już ugruntowanych jest znaczna dysproporcja co do ilości otrzymanego przysłowiowego "blasku reflektorów" badań.

Cel: Cel (#1): Celem tej pracy jest stworzenie i udostępnienie współdzielonego, ogólnodostępnego *Katalogu Zapachu Kodu* z prostą metodą kontrybucji nowych treści, tak aby technikalia nie stanowiły przeszkody w dodawaniu nowych informacji. Katalog ten ma pełnić funkcje zarówno przydatne dla badaczy – jako zunifikowany, jednolity system danych, umożliwiający natychmiastowe wydobycie informacji, ale także dla programistów o szerokim zakresie doświadczenia – dla juniorów, jako źródło wiedzy, jak i dla tych bardziej doświadczonych, jako zestaw praktyk, do którego można odesłać podczas dyskusji lub sporu podczas systematycznego przeglądania kodu. *Cel (#2):* Ponadto celem jest zidentyfikowanie wszystkich możliwych pojęć, które można scharakteryzować jako Brzydki Zapach Kodu, a także wszelkie kontrowersje, jakie istnieją w tym temacie. *Cel (#3):* Na koniec - scharakteryzowanie zapachów kodu poprzez przypisanie im odpowiednich cech wynikających z głównych, już istniejących w literaturze taksonomii.

Metodyka: Przeszukałem zarówno literaturę, w której brzydkie zapachy kody są już zaklasyfikowane jako takowe (innymi słowy, brzydkie zapachy kodu wyszukiwalne przy pomocy zapytań frazami wyszukiwania w tematyce `code smell`), ale także miejsca w literaturze, które mówia o pewnej koncepcji pasującej do definicji brzydkiego zapachu kodu, ale jednocześnie nie będąc z tą frazą nigdy powiązana. Przeszukałem również "szare dane", w tym: fora dyskusyjne, artykuły techniczne, społeczności programistów, a także blogi i strony domowe programistów; zarówno w celu wyodrębnienia potencjalnych nowych zapachów kodu, jak i zidentyfikowania i rozwiązania wszelkich istniejących kontrowersji związanych z istniejącymi zapachami kodu. Przeanalizowałam wyniki i zinterpretowałam je wykorzystując wiedzę zgromadzoną w źródłach literaturowych, klasyfikując je i weryfikując ich spójność wewnętrzną.

Rezultaty: Zidentyfikowałem 56 zapachów kodu, z czego 16 to nowe, oryginalne propozycje, oparte na jeszcze nie powiązanych zagadnieniach z literatury, "szarym źródle danych" oraz osobistych doświadczeniach branżowych. Dwie dotychczasowe koncepcje zapachów kodu, które wzbudziły największe i uzasadnione kontrowersje, zostały zmodyfikowane i dostosowane do obecnego stanu wiedzy. Stworzyłem katalog internetowy, w którym istnieje możliwość filtrowania bazując m.in. trzech taksonomiach. Katalog ten również zawiera pojedyncze artykuły dla każdego ze zidentyfikowanych Code Smelli. Każdy Code Smell otrzymał klasyfikację ze względu na taksonomię, synonimy (bądź zapachy blisko spokrewnione), metody ich refaktoryzacji, rodzaj problemów jakie sprawiają (w tym: jakie pryncypia programowania łamie), relację w stosunku do innych Code Smelli, notkę historyczną oraz autorskie minimalne przykłady "śmierdzących" implementacji oraz ich solucji.

Konkluzja: Temat brzydkich zapachów kodu jest bardzo wartościowy i powinien być istotny dla każdej osoby w branży: deweloperów (efektywność), właścicieli produktów (jakość), analityków (dług techiczny, tempo wdrażania) i klientów (wsparcie oraz rozwój), nie mniej jednak, nie ma on szeroko przetartych szlaków w powszechnej świadomości. Ta praca może przyczynić się do zwiększenia świadomości w branży, a także wyregulować braki w istniejącej literaturze naukowej. Pozwoli na podejmowanie dalszych badań w sposób świadomy, gdyż dostęp do skumulowanego zasobu informacji dzięki katalogowi, nie jest już ukryty lub utrudniony. Zunifikowane dane pozwolą na lepszą reprodukowalność badań, a kolejne otrzymane wyniki mogą być bardziej definitywne.

1. INTRODUCTION

The number of new developers grows proportionally to the 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 outside of the code perspective. 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. This realization made me unable to continue with my initial Master's Thesis. One of the latest meta-analyses confirmed my thoughts, which has reached the same conclusions. 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 web-building technical knowledge, focusing on the merits.

In this thesis, I will 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 research itself but also from the gray data. I will try to find any controversy and suggest alternatives for them. I will also add completely new concepts classified as Code Smells, both resulting from my 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, I also scour the Internet, including message boards, developers' home pages, and any other source containing substantive three cents on this topic.

1.1. ORIGINS

"Poor solutions to recurring implementation and design problems" [59]. This statement is the genesis of what I am discussing in this master thesis - the *Code Smells*. As Fowler put it, in 1999, in his book called *"Refactoring: Improving the Design of Existing Code"* in the chapter with the same name as the discussed issue: *"code smell is a surface indication that usually corresponds to a deeper problem in the system"* [24]. The origin of the name *"Code Smells"* has an amusing story with a hilarious interlude and a good allegory that builds an intuitive understanding of the issue and highlights an essential adverb in the definition. Fowler had no problem explaining how to *"refactor"* a particular code, as it is straightforward. However, he found an issue specifying **when** exactly such refactor has to be done. The author believes that it is not apparent but somewhat vague. At that time, when he was mulling about it, he was visiting his friend Beck. As he described, his friend suggested the *Code Smell* name, most likely inspired by the "unfavorable" smells generated by his newborn daughter at that time [56].

1.2. MEANING

Code Smell by itself is not a defect or - as it has been the common saying for a long time in the field of software development - a bug. It does not break the functionality of the software by any means. In the extreme case, the entire application could be filled with as many code smells as possible while still being fully functional and working correctly. In the worst-case scenario, valid code can be transformed through an obfuscator, making it utterly unreadable without a slight chance of comprehending anything. After such an obfuscation operation, we do not lose reliability *(if the obfuscator was reliable)*, but we destroy the code quality, which is precisely what this term is about [61, 77].

The word "when" in the preceding section is bold for two reasons. A code smell does not imply that something should be done about it immediately. If there is a piece of code somewhere in the corner of a company that does the same minor process job and is not changed for years, but now, due to the software decay process or the gained experience, someone realizes that it is smelly and that it does not need to be refactored. Second, the question itself of whether something is smelly is debatable [56].

One might ask himself why anyone should be interested in this? If something works, why bother with further inspection? Using business jargon: The project is completed, and the client is satisfied. The answer to this question comes from the aforementioned client who requests a new change or some other new functionality, but the code does not allow us to add the desired thing easily. In addition, after time, the entire team may even have a problem with the implementation of the functionalities but even with understanding how exactly the code works.

1.3. IMPLICATIONS

Now, here the problem can finally be quite apparent as the implications of low-quality code extend on the reusability of any components for any future projects and plans, extendability of the written software in case of any new requests, understandability of the code if anything has to be re-examined, the flexibility of it, effectiveness and functionality [8]. Considering that software projects can go on for years and decades, not only in the maintenance phase but also in the development phase, assessing the code quality is crucial.

Lindy's Law is worth mentioning described by the mathematician Eliazar: "What has been around the longest is likely to remain around the longest" [20]. We can translate that into "the longer a technology is, the longer it is likely to remain" in our technology realm. This implies that all the programming languages that we know throughout generations, like *C* or *Java*, are very likely here to stay for quite some time, and thus, this is why we should pay attention to the quality of the code we write, as it is likely here to stay with them too.

However, what if someone does not agree with this concept? He will decide to "just do the job", not minding how dirty it might be, and as long as the mechanisms work together, he will be satisfied. It does not matter whether the whole system is cleverly arranged or neatly tidied up. For him, the ubiquitous duct tape or zap straps that hold everything together are splendid. This approach can be called a *Technical Debt* maximization strategy, which is not a very sane approach in the tactical field of software architecture.

Technical Debt (also known as *code debt* or *design debt*) is a metaphor that tells of a collection of all activities that have been postponed to provide new functionalities instead, thus, focussing on short-term reward rather than long-term maintainability. This debt is intended to be paid off in an uncertain future (if at all) [13]. It can be understood as a concept that addresses the potential future cost (which, depending on the strategy, must or may not be paid back) due to the implementation of easy or limited solutions in the present. The lack of bothering about the Code Smells can be in the set of such debts, next to technology choices, libraries, programming language, or lack of software updates [98].

Following the previously mentioned "strategy", along with the subsequent unreasonable implementations of new functionalities that generate Code Smells, the implementation of new ones will require increasingly more time devoted to their implementation due to much more difficult reading, searching, and navigating through the repository [13, 48, 84]. With each such implementation, the project's complexity grows exponentially, which will sooner or later lead to a level where new team members will not be able to contribute anything meaningful. Even the maintenance effort increases. The cost of maintaining such a project increases, and attempts to approach the code to become more complex [49, 80]. At some point, it will block the project to such an extent that the most efficient approach is its complete reimplementation from the very beginning.

Unfortunately, we do not live in a perfectly isolated world. I assumed that a person who does not care about code quality would create software that would work flawlessly. This result of work is improbable to happen. In other words, Code Smells are highly correlated with error-prone software [14, 34, 65, 67]. Of course, in practice, there is also no question of creating a more extensive project; we even have a saying for it: a giant with feet of clay.

In the literature, Code Smells are used as an attribute (metric) that measures the amount of Technical Debt in a given project, for example, by referring to *SonarQube* as the Technical Debt measurement tool [46]. Code Smells are one of the direct causes of increasing Technical Debt, and Technical Debt is already a well-established term that resonates with businesses and analysts, as proper management of this factor is crucial for business success and product quality [96].

1.4. CAUSATIONS

It is not difficult to develop a shortlist of potential reasons for poor quality code. They can come from various sources and during various stages of application development. For example, the very first smelly piece of the puzzle could already be located during the initialization of the project, after the application design process, where the use of a mismatched architectural design pattern could be established [27].

Probably the most intuitive causation would be the inexperience of developers [17]. If the developer has not had the opportunity to work on a few medium- or large-scale projects, he may not know how best to tie code structures together to avoid making things more complicated than they should be [10]. For the same reason, individual methods, functions, or classes cannot be implemented appropriately.

Exactly to counter this, there are code reviews in which developers can mutually suggest better solutions and thus also learn the profession. Of course, this serves novice developers and the whole range of them, regardless of experience: There is no infallible person, and each additional pair of eyes increases the chances of getting the most optimal code. Hence, the lack of such a process may be another reason for the project codebase's rising number of code smells.

However, the above is just an assumption. The code review could work efficiently and strain part of the smells, but the catch is that developers would have to be aware of the existence of Code Smells. Unfortunately, based on 2013 surveys, we know that 32% have never heard of it, and another 50% have admittedly heard of it but never got interested in it [94]. To work smoothly, developers should know more about Code Smells and be aware of them instead of how it is now [69, 93].

The core of the problems does not have to be restricted strictly to the development and design processes. External factors can also cause them. If the developer has to deliver his solutions in a limited time frame, he often has to take shortcuts that - consciously or not - create Code Smells. Pressure from the business side is the most frequently mentioned causation in the discussion forums, but it is also closely investigated under the term Technical Debt [57].

In the search for code smell causation, we can also delve into the peace of mind or developers' emotions. Numerous studies show that developer productivity is closely related to their mental well-being: The better a developer feels, the better his performance, and the worse his mood, the more mistakes he makes [31, 32, 33, 64]. I have not found any research that studies the influence of mood precisely under the code smell terms, but it may reasonably be concluded that a similar correlation is most likely observable.

Surprisingly, however, the topic of code smell causation has not been thoroughly examined from the focused research point of view. No research has been conducted to identify the root causes of Code Smells [35].

1.5. TOPIC EXPANSION

1.5.1. Code Smells and Taxonomies

Going back to the aforementioned Fowler's book, this was when the concept of Code Smells was born. He listed 22 different Code Smells in an unstructured list whose names have been well established in research and remain unchanged.

Several years later, in 2003, Mäntylä took the next big step in his master thesis [51]. He created the first taxonomy that also established its position and remains in research today. He took Fowler's list and divided it into seven groups that I would call a breakdown based on the type of obstacle (obstruction) they create. It was an excellent idea, as it was the first significant step towards improving the understandability of the assumptions and connections between Code Smells. Before, they were just intuitive hints based on which one would have to use a given refactoring method. This has opened the door to further research.

Since then, there have been three more major works in which new Code Smells have been defined: Wake's 2004 refactoring book [92], Karievskii's 2005 refactoring book [43], and Martin's 2006 book "Clean Code" [55]. New Code Smells were extracted from these sources, which also appear more or less frequently in the literature.

Unfortunately, the research is disproportionate. The Code Smells defined by Fowler gained much more attention than those defined by the previously mentioned authors, and the disproportion has not evened out over the time available for research.

Regarding the development of taxonomy, there are two more suggestions here. Wake structured the Code Smells into named chapters, which I define as grouping by place of occurrence. Unfortunately, this grouping did not gain noticeable traction in the research area. This may be because it was never strictly written as an open proposition. As a bonus, Wake also included not only his new propositions in this list and Fowler's ones, so for 2004, it was a complete, orderly proposition.

The third taxonomy that I have found is the proposal to group Code Smells according to whether they appear within a given class or whether the problem spreads between classes. Looking at the so-called "grey knowledge", the first proposal of this kind can be observed in 2006 in Jeff Atwood's "Coding Horror" webpage [6]. In literature, it first appeared in 2013 in the paper by Marticorena, titled suggestively "Extending a Taxonomy of Bad Code Smells with Metrics" [54]. They took the Fowlers list and, in addition to the Jeff Atwoods propositions, created additional Boolean-like attributes based on inheritance, access modifiers, and granularity. This study has received some traction since it has 25 citations at the time of writing (for comparison: the average is 64.2 citations per Code Smell study, while the median is equal to 9 [44]).

One more remark is that in their work, they have also added the Mäntylä division, but without the *Encapsulators* group, dropping the "Middle Man" and "Message Chain" into the *Couplers* group. Mäntylä himself wrote that it is debatable whether these two smells should be in the category *Encapsulators* or whether they are placed in *Couplers* and *Object-Oriented Abusers*, respectively. In the Mäntylä paper of 2006, we can see that he eventually placed these two smells in the *Couplers* category. This may mean that these "smell drawers" are not entirely clear - they can intertwine or adapt depending on the research needs [53].

1.5.2. Bad Smells - The Hierarchies

Apart from Code Smells, the term *smell* itself, referring to things that "seem like something is wrong", has gained substantial momentum. Beyond the Code Smells, the general term Bad Smells has emerged, as Van Deursen in 2001 suggested, that bad code smells were not limited to the source code itself [89] - however, it is not clearly defined. Currently, the term does not mean some superset in which one of its elements would be a set of Code Smells. It is used more for the lyrical variety (as a "permutation" of *Bad X Smell*) as a synonym, not constantly using the phrase Code Smell throughout the paper.

I mentioned these collections because now we have more than just the Code Smells issue in the literature. Entirely new categories have emerged to examine similar problems from different perspectives. One such example would be Design Smells, which looks at the system from the structural view. Unfortunately, it seems that despite their large number, only the Code Smells and Design Smells are blazing the trail in literature, and the rest are novelties that are only rarely mentioned. It does not change that the researchers are trying to find new spaces in which suspicious things may appear anyway. I have managed to reach 22 such hierarchies (in addition to Code Smells) in the field of software engineering, both in the literature and on the researcher home pages:

1. *Architectural Smells*
   - Set of architectural design decisions that negatively impact system lifecycle properties (understandability, extensibility, reusability, testability) [29].
2. *Design Smells*
   - Recurring poor design choices [86].
3. *Implementation Smells*
   - Subset of Code Smells that has the "within" expanse attribute and other specific granularity based on Sharma *House of Cards* paper from 2017 [76].
4. *Comments Smells*
   - Comments that can degrade software quality or comments that do not help readers much in terms of code comprehension [39, 55].
5. *Linguistic Smells*
   - Smells related to inconsistencies between method signatures, documentation, and behavior and between attribute names, types, and comments [5].

6. *Energy Smells*
    - Implementation choices that make the software execution less energy efficient [91].
7. *Performance Smells*
    - Common performance mistakes found in software architectures and designs [81, 82].
8. *Test Smells*
    - Poorly defined tests; their presence negatively affects comprehension and maintenance of test suites [88].
9. *UML Smells*
    - Model smells and model refactorings applicable in the early stage of model-based software development that violate its 6C Goals - Correctness, Completeness, Consistency, Comprehensibility, Confinement, Changeability [4].
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 [18].
11. *Community Smells*
    - Sub-optimal organizational and socio-technical patterns in the organizational structure of the software community [68].
12. *Bug Tracking Process Smells*
    - Set of deviations from the best practices that developers follow throughout the bug tracking process [70].
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) [75].
14. *Environment Smells*
    - Smells that make work less comfortable by, for example, requiring more steps than it should be to achieve specific actions [55].
15. *Presentation Smells*
    - Guidelines to create better presentations [74].
16. *Spreadsheet Smells*
    - Intra-Code Smells but for Worksheet End-User Programmers [37].
17. *Database Smells*
    - Antipatterns in terms of logical database design, physical database design, queries, and application development [40].
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 [3].
19. *Android Smells*

    - Violation of standard principles and practices that have an impact on the quality, performance, comprehension, and maintenance of mobile applications [11].
20. *Security Smells*
    - Security mistakes that may jeopardize the security and privacy, identification of avoidable vulnerabilities [30].
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) [85] 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*.

1.5.3. Focus of Research

Moving back to the subject of Code Smells, the current research focuses on answering various exploratory, base-rate, relationship and causality questions [44]:

1. *Exploratory*
    - Majority of Research.
    - Includes:
        - Questions about the *Existence* of particular Code Smells (32,5%).
        - Attempts at *Description and Classification* of Code Smells (75%).
        - *Description-Comparative* analyzing the differences between pair of Code Smells (17,5%).
2. *Base-Rate*
    - Majority of Research.
    - Includes:
        - Questions about the *Frequency Distribution* of Code Smells (27,5%).
        - *Descriptive-Process* of examining Code Smells (25%).
3. *Relationship*

- Minority of Research.
- Defining the *Relationships* (i.e. correlations) between pairs of Code Smells (30%).

4. *Causality*
   - Minority of Research.
   - Includes:
     - Whether a Code Smell *causes* another Code Smell (22,5%).
     - *Causality-Comparative* - the comparative magnitude of causation of a Code Smell between two selected Smells (12,5%).

1.6. PROBLEMS

Reading this thesis so far, as much as I would like to hide it just yet, you probably already could see a noticeable lack of order and standardization in the literature investigation of Code Smells. There are several significant problems, and it is hard to tell which is the most concerning. Without paying much attention to the order, let us begin the elaboration.

1.6.1. Lack of Standardization: Particular Code Smell Definitions

There are many smells under different names that cover more or less the very same topic. In the vast majority of the literature, when considering a single study, one name is usually selected and used from the beginning to the end (without mentioning synonymous names or explanatory definitions). Given, for example, the most enigmatic set of names: *(Large Class, Brain Class, Complex Class, God Class, God Object, Schizophrenic Class, Ice Berg Class, Blob)*, it is hard to differentiate them one from another even after some time spent searching for distinctions between them (as they differ even in the "differences" between papers). Some of them name the same thing, some are only slightly different from each other, and some use slightly different predictors.

The Tertiary Systematic Review paper from 2020 concludes the same observation in their *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"*** [44].

It should also be noted that there is no breakthrough on the revised names of the smells. Fowler, in 2018, changed the name of some of his smells, such as *Inappropriate Intimacy* to *Insider Trading*, or generalized the concept *of Lazy Class* to *Lazy Element*, which is noticed only in the few most in-depth studies.

1.6.2. Lack of Standardization: Particular Code Smell Categorization

The taxonomy proposed by Mäntylä in his 2003 thesis is the most widely distributed and most often (if not exclusively) used. Its division is not carved in stone; as Mäntylä himself mentions when defining them, whether the code smells from the *Couplers* category should be included in the *Object-Oriented Abusers* category is a moot point. Then there are the smells from the *Encapsulators*, which could be spread over *Couplers* and *Object Orientation Abusers*. The central theme of the segregation, as he mentions, is to make the Code Smells list more intuitive by categorizing it, but it turns out that everyone understands the smells a bit differently, which is why they are ultimately not unequivocally assigned. I elaborate more on that in the literature review section [ref. Section 2.4] as well as the entire 3rd chapter [ref. Chapter 3].

1.6.3. Lack of Standardization: Bad Smells and Antipatterns

Another thing is that there is no demarcation 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*). When examining this issue, I was unable to determine at first whether a *Feature Envy* is defined as a Code Smell [56] and only a Code Smell or should it be a Design Smell [2] and, therefore, whether it should be only a Design Smell or should it be both. Then it came up as *Bad Smell* [21], and so on. This lack of constraints 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 very diminutive [78]), but in turn, some Bad Smells are designated as Antipatterns, and the other way round. This hampers targeted research; Researchers are forced to include all these concepts in their search string queries, as in [listing: Listing 1.1] [47]. 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" [60].

Listing 1.1. 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 (
    ...
)
```

1.6.4. Lack of Go-To Source

One of the main reasons for this lack of standardization is the lack of a common source that would hold the latest aggregated information. Not even one that could be used as a research data set, but at least one that could be used as a reference guide to determine everything one may need to know in the Code Smells area.

Outside the literature, on the Internet, after performing a simple search for *code smells* (from the private session), among the things that a person looking for information could typically click on is a:

1. Short Fowler blog post elaborating on Code Smell from 2003, a very short article explaining Code Smell as a glossary term.
2. Selectively descriptive Wikipedia page.
3. Longer article with Code Smell explanation on the Mäntylä taxonomy level.
4. A concise article explaining Code Smell as a glossary term, containing one-sentence explanations of three Code Smell taxonomies with misleading information.
5. Two most extensive resources:
   - Atwood's blog post about Code Smells from 2006, that contains a subset list of them (as defined by Fowler, Wake, and Karievsky) grouped into smells occurring within and between classes and their short descriptions.
   - Dedicated web page[1], which provides the Code Smells (those defined by 1999 Fowler) explanations and refactoring methods, structured into the Mäntylä 2006 Taxonomy (without *Encapsulators*) with unknown additional pay-wall content.

There is only one open-source place that one can go to (Wikipedia). There is no single place that contains updated information on Code Smells. At the moment, in order for someone to obtain the latest information about the current stance of Code Smells, he would have to refer to the latest meta-analyses, which, without intuitive knowledge or "nutshell" brief explanations at hand, can be confusing with the number of terms and the lack of a

[1] Refactoring Guru - `https://refactoring.guru/` (*available: 15.03.2022*)

specific structure explaining the issue. This is understandable since that is not the purpose of these meta-analyses either. What is more, the very fact that the meta-analysis concludes that the information on Code Smells, at the present moment, is scattered and mixed up can lead the person eager to learn to give up and abandon further analysis.

This lack of a good source for up-to-date information contributes to the previously mentioned issue of lack of awareness and knowledge about the subject among developers [Section 1.4].

1.6.5. No further new Code Smells contributions

Since Kerensky's book (2005) [43], there were no significant contributions to the list of known Code Smells until the update of three smells in Fowler's third edition of his refactoring book in 2018 [26]. All contributions to the Code Smells list appeared only in the books if we do not count the variations based on the already defined smell (e.g., *Large Class*) or in entirely new hierarchies of smells. It seems that proposing a new Code Smells requires knowledge derived from practical experience, observation skills, and creativity. However, I would like to note that these requirements and book-writing are not the only possible way of introducing new Code Smells.

Performing a literature review in a systematic review manner, with thoughtful predefined queries, has a wide range of benefits that cannot be disregarded, but there is a small catch. I believe that there are things that fit into what we describe as Code Smells but still, so far, have not appeared among the names that would be caught during typical Code Smell research (i.e., to put it simply, they did not appear in the context of Code Smells and were not associated with them). One such example that I have found is *Boolean Blindness* [16, 90, 95], the first appearance of which is dated to 2011 by Licata, as told by Harper [36]. In addition, discussion forums, popular science articles, and technical support forums can be sources from which new Code Smells can be extracted as recognized by the developers' community. In this thesis, I contribute a widely-known example, *Callback Hell*.

1.6.6. Summary

It is worth summarizing all the above observations. They have been grouped into three of the most important general points.

1. There is a standardization problem:
   - Particular Code Smells:
     - Sometimes, the same Code Smell is understood differently.
     - The same Code Smell regarding the same issue might be in the literature under different smell names.
     - There is a lack of a go-to place for each of the smells.
   - Bad Smells concepts:

    - The concept of Code Smell is often redefined, and its meaning is very vague at this point.
    - The term Bad Smell is often mixed with the term Antipattern. There are three points of view: synonymous use, noticing a subtle difference & noticing clear distinction.
  - Taxonomies:
    - Taxonomies are used differently, if even at all.
    - Even when the same taxonomy is used, different versions are used.
    - Even when the same taxonomy is used in the same version, they differ in the permutation of Code Smells placement in its categories.
2. There is an awareness problem:
  - Among Developers:
    - 32% of developers are not aware that there is such a thing as the Code Smell topic.
    - 50% of developers never looked deeper into the Code Smell topic, just heard about it.
    - There is a lack of a go-to place for each of the smells.
  - Among Researchers:
    - There is a disproportion in research on individual Code Smells - some are extensively studied and others very rarely or not at all.
    - There is no more significant consideration of the existence of more Bad Smells hierarchies.
    - It is possible that the potential to use Bad Smell as an umbrella term for all the types of smell hierarchies is being wasted in favor of using a synonym or abbreviation term.
3. There is a definition problem:
  - Some Code Smell names are superficially misleading and controversial. This may result in a reluctance to delve into the concept (given the times when the quality of information is judged after the first one-two sentences).
  - There is no ongoing research that would extract more Code Smells from software engineering literature, which is not regarded as Code Smells yet (searching for issues that so far have not stood next to the keywords "bad smell" or "antipattern").

1.7. THESIS GOALS

The thesis aims to address the problems mentioned in the previous Section 1.6 and provide a solution that addresses them. That is a development of a webpage catalog that can hold various data about Code Smells and would act as a unified source of information. Ideally, this catalog should fulfill the functional assumptions of the easy contribution of new content, being a unified source of data with the possibility of easy extraction, and an encouraging presentation for people interested in learning about the subject of Code Smells.

The second goal is to find and collect the data to fill out the substantive side of the catalog. First of all, supplemented with all the data on Code Smells, which has already been mentioned in the literature under this term strictly. Secondly, with the data from the literature that talks about the topic of Code Smell, but has not been recognized as such, thus making it currently impossible to discover with Code Smell as a keyword. Lastly, adding entirely new Code Smells based on grey data, including home pages of specialists and authorities, discussion forums, and other sources, processed and filtered based on domain knowledge and experience.

The third and last main goal is to classify and assign the attributes to all of the Code Smells.

1.8. CONTRIBUTIONS

First and foremost, a web directory is attached to the paper, which stores the information about all the Code Smells mentioned throughout this thesis. Contrary to just creating a new article with the systematic literature review formula or meta-analysis which would eventually become obsolete as the years go by, leading to the same problem, **this solution anticipates that and prevents the recursive emergence of the problem in the future**. The catalog is interactive and open to the public, both from the convenient overview and the source code. Due to the merits of the matter, **a research paper was also prepared which was submitted for publication as a chapter in a book to be published by Springer**. Besides that, from the substantive side, I have found and listed 67 Code Smells in one place which has not been done so far and significantly facilitates access to information. Moreover, I collected the attributes and characteristics of each of them, organized them according to my taxonomic classifications inspired by the existing ones, and completed them. What is more, I created their short descriptions, sources, and sample source codes along with potential solutions. I also proposed a new concept of "hierarchies" that would systematize the concept of Bad Smells, and I listed 22 such hierarchies that can be found in the current literature. It is also completely novel, and the proposal of such a perspective will significantly facilitate the understanding of a topic that is currently somewhat fuzzy due to the lack of a go-to summary resource from which one could grasp an intuitive understanding of this subject.

1.9. THESIS SCOPE

In this thesis, in the following sections, I expand more on the findings and conclusions that could be read in the current chapter, sourcing them from both the literature and the information available on the Internet discussion forums, article sites, and personal blog posts. That research was conducted not only about Code Smells themselves but also among the concepts that could be defined as Code Smells but were not used in tandem with this expression (e.g., `"common problems code"`, `"bad solution programming software"`, etc.) in Chapter 2.

In the Chapter 3, I elaborate a little more on the topics of existing detected taxonomies and summarize them with a unified summary proposal. There are tables sorted by the chronological occurrence of a given Code Smell (as a Code Smell concept) and by the assignment to all taxonomies.

Chapter 4 contains a complete list of Code Smells that I managed to reach or define myself. For each Code Smell, I list all its potential synonymous names that have appeared in the literature, unifying them. In addition, I assign the taxonomies for each of the Code Smells, from which other Bad Smell hierarchies can be considered, its historical origin, an explanatory description, and the simplest, minimal code example and its solution.

Along with this thesis, an open-source, public *Code Smells Catalog* is provided. When developing this tool, I focused on three aspects. First of all, all added content should fulfill the role of both a unified form of data for research purposes and provide an educational and informational value for everyone. Second, due to a large number of Code Smells, I wanted to make the website able to filter the content, thus making it easier to navigate through it. Finally, I ensured to implement the tool in such a way so that the knowledge threshold for each willing content contributor is as low as possible. Chapter 6 covers this in detail.

2. LITERATURE REVIEW

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 [44]. 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

To not leave this thesis "dry", I will present a brief historical outline. Before my writing work began, I planned to do the Master Thesis under the "Code Smells Prediction" title, but things have changed over time. When conducting a systematic review to investigate more on the topic, I have noticed that the current literature is missing out on, consciously or not, some of the issues related to Code Smells. I further examined it, searching for more information, and noticed more shortcomings. I decided to pursue the topic of this thesis, putting away my previous research and starting a deep investigation for information. Therefore, I define five research questions.

1. ***RQ1****: Do all Code Smells have similar amount of research devoted to them?*
   - A holistic approach would be preferred to ensure that nothing escapes a potential classification during testing. Not taking all factors into account can lead to differences in research results.
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 (their 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 "gray 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?

Please note that because of the reasons why I started my research, I can be influenced by confirmation bias or selection bias. I have also excluded studies that are not published in English. This fact should be borne in mind when reading the work, and in case of finding any problems, be sure to question and challenge them.

2.2. SOURCES OF RESEARCH

The limiting factor in conducting a fully-fledged systematic review for any paper is the number of people conducting the research and available time. Thus, in a rapid review fashion, *Scopus* was used as the primary source of research using search strings. Furthermore, I 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*, etc.), blog entries of various field experts, courses, and guides prepared by respected IT authorities, as well as websites and videos devoted to Code Smells. I 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.

To illustrate the numerical values (and answer **RQ1**), I collected very general information on the amount of searchable research in the context of Code Smells on each individual Code Smell, with a simple search query in Listing 2.1 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. This action imitates a quick look-up as if someone hearing about the topic wanted to quickly get more information. Please note that the latter part of the query might be different based on other, synonymous terms for given smells. I 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, I 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 2.1. Search String Queries Structures

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

2.3. SEARCH QUERY RESULTS

I have aggregated the results from the search queries in Listing 2.1 into a sorted table Table 2.1, 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. Either way, 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.

2.4. LITERATURE REVIEW AND GENERAL INVESTIGATION

After reading the data and the information that I 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 in 2003 - from his Code Smell Taxonomy article from 2003 [52] that features seven groups: *Bloaters*, *Object-Oriented Abusers*, *Change Preventers*, *Dispensables*, *Encapsulators*, *Couplers*, and *Others*. He adjusted his work in 2006 [63] when he moved *Parallel Inheritance Hierarchies* into the category of *Change Preventers*. Despite that, studies place particular smells in different categories (the article from 2018, where *Parallel Inheritance Hierarchies* is once again labeled in *Object-Oriented Abusers*

[35]). 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, I would consider it as the 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". I elaborate further on mixing inside a taxonomy in the section dedicated just to that matter in Section 3.3. This already indicates the answer to **RQ3**, together with the information contained in the *Problems* section of the *Background* Section 1.6.

Some studies are focusing more globally on inspecting all smells as a whole, in different contexts such as the evolution of smells [72], their effects [73], 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* [72], making it harder and harder to pinpoint everything by a predefined search query; more about that in the *Bad Smell Hierarchies* Section 3.2. Narrowing the Perspective to Individual Code Smells, *Large Class* has the most subvariations (*Brain Class*, *Complex Class*, *God Class*, *Schizophrenic Class* [22], *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. Answering **RQ5**: These subvariations of existing smells are the only new Code Smells that appear in the literature Section 1.6.1, but speaking of Bad Smells in general - numerous new hierarchies are created that define completely new smells.

The most recent comprehensive study that aggregates Smells comes from 2020 [44], where the authors conducted a systematic review with a great amount 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 [54] (or, outside the scientific literature, the Atwood taxonomy). 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 is fine, 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 additionally - in terms of 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 regarding the 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 the creation of a tool for standardization purposes, insisting on another study they investigated that suggests the creation of something like a Code Smell Catalog** that I started to create myself, even before I reached the paragraph for the very same reasons and conclusions.

Thus far, we see that the Code Smells are disproportionately investigated. Some of them are entirely omitted. The various smell hierarchies (*design smells, code smells; refer to Section 3.2*) occasionally intertwine smoothly without distinctions. But what about Code Smells' discussions (**RQ4**)? The study of 2021 [79] mentions, *"for Data Class various exceptions to the definition are discussed, which are 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 [99] 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 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 [43] up to the updated book by Fowler in 2018 [26]. This absence 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 community insights and ideas, and no search outside the *Code Smell* as a phrase to find existing phrases that fit into 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 future problems with comprehensibility, readability, maintainability, or extendability. This impression is incorrect. I have found one more Code

Smell *Tramp Data* that was mentioned in a book by Steve McConnell, *"Code Complete"*, before the existence of the term *Code Smell*, which is also recalled in the "gray data source" by Steve Smith in his *"Refactoring Course"* from 2013. He also mentions six more code smells absent in the scientific literature. I 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, I have defined 15 more Code Smells, but more on that is given in the Code Smell List Chapter 4.

Code Smell	**Num#**	**Code Smell**	**Num#**
Data Class	379	Inconsistent Style	2
Large Class	370	Incomplete Library Class	2
Long Method	187	Inappropriate Intimacy	2
Feature Envy	129	Inappropriate Static / Static Cling	2
Regions*	89	Base Class 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 / Impure Functions	45	Tramp Data	1
Explicitly Indexed Loops*	36	Null Check	1
Refused Bequest	27	Binary Operator In Name	1
Dead Code	23	Afraid to Fail	0
Message Chains	18	Required Setup or Teardown Code	0
Parallel Inheritance Hierarchies	14	Indecent Exposure	0
Long Parameter List	13	Insider Trading	0
Hidden Dependencies	11	Uncommunicative Names	0
Conditional Complexity	9	Explicitly 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	Complicated Boolean Expression	0
Magic Numbers	4	Complicated Regex Expression	0
Inconsistent Names	4		
Temporary Field	3		
Lazy Element	3		
Inconsistent Abstraction Levels	2	**Code Smell****	**1555**

Regions & Loops** - these terms could give a lot of false-positives.
*Code Smell*** - number of results for a query without any particular Code Smell Name

Table 2.1. Results of Code Smell Search Queries in Scopus

3. CODE SMELLS TAXONOMIES AND BAD SMELL HIERARCHIES

To fully understand current knowledge about the Code Smells, it is worth getting acquainted with the taxonomy of this issue first. I gathered information from various sources to summarize the possible attributes that may be characteristic of Code Smells. I was unable to find a publication that summarized the Code Smells themselves and **aggregated** the groupings of the Code Smells. The information is scattered, and having more perspectives is undoubtedly beneficial, so in this chapter, I summarized them and proposed distinct names for each of them. In addition to that, I also performed a literature review to determine how exactly prevalent smells are in research papers.

3.1. DIFFERENT GROUPING OF CODE SMELLS

3.1.1. Mäntylä Obstruction Taxonomy

Mäntylä, in his Master Thesis [51], classified the Code Smells given by Fowler in 1999 into six different significant groups to increase the understandability of code smells as a concept. This grouping constitutes a standard division into subcategories: Bloaters, Object-Oriented Abusers, Change Preventers, Dispensables, Encapsulators, Couplers, and lastly, Others - presented in Table 3.1.

Nowadays, this is the most common taxonomy used in Code Smells research, although this list currently appears in various permutations and variations. Using two examples, *Message Chain* and *Middle Man* are sometimes moved out of the *Encapsulators* category, completely removing this category, while smells are moved to *Dispensables* or *Object Oriented Abusers*. Mäntylä himself made this change in his article from 2004, *"A Taxonomy for „Bad Code Smells”"* [63]. In another Mäntylä paper from 2006, *Parallel Inheritance Hierarchies* is placed in the *Change Preventers* category (visualized in Table 3.2), while mentioning that it might also be placed in *Dispensables*. Marticorena follows both of these permutation changes in code smells in his 2006 paper *"Extending Taxonomy of Bad Code Smells with Metrics"* [54].

There is no final state of this taxonomy or in what permutation it shall be used. A systematic review from 2018 on smell detection lists six groups [72]:

— The Bloaters,
— The Object-Oriented Abusers,

— The Encapsulators,
— The Couplers,
— The Design Rule Abusers,
— The Lexical Abusers.

Resigning from the bijection relation of groups with smells, summarizing that, for example, Message Chain is placed in both The Couplers and The Encapsulators [72].

Group Name	**Code Smell**
The Bloaters	Long Method
	Large Class
	Primitive Obsession
	Long Parameter List
	Data Clumps
The Object-Oriented Abusers	Switch Statements
	Temporary Fields
	Refused Bequest
	Alternative Classes with Different Interfaces
	Parallel Inheritance Hierarchies
The Change Preventers	Divergent Change
	Shotgun Surgery
The Dispensables	Lazy Class
	Data Class
	Duplicate Code
	Speculative Generality
	Dead Code
The Encapsulators	Message Chain
	Middle Man
The Couplers	Feature Envy
	Inappripriate Intimacy
Others	Comments
	Incomplete Library Class

Table 3.1. Mäntylä Taxonomy (2003) [52]

I call this type of taxonomy *obstruction grouping* - the names define what kind of the primary obstruction the smells cause. Thus, in an example, *Large Class* and *Long Method* bloat the codebase, *Feature Envy*, and *Inappropriate Intimacy* are coupling things together and *Refused Bequest* or *Switch Statements* are abusing object-oriented programming.

As mentioned above, this obstruction grouping is most often mentioned in the papers, but due to the lack of standardization and the lack of a classification of new smells in this category, various new subcategories are proposed or removed. The most definitive list, supplemented with new code smells, might be the one proposed by Steve Smith in his *Refactoring: Fundamentals Course*. It is based solely on its publicly available table of contents, in which he classifies smells with one additional subcategory: *Obfuscators*.

Group Name	**Code Smell**
The Bloaters	Long Method
	Large Class
	Primitive Obsession
	Long Parameter List
	Data Clumps
The Object-Oriented Abusers	Switch Statements
	Temporary Fields
	Refused Bequest
	Alternative Classes with Different Interfaces
The Change Preventers	Parallel Inheritance Hierarchies
	Divergent Change
	Shotgun Surgery
The Dispensables	Lazy Class
	Data Class
	Duplicate Code
	Speculative Generality
	Dead Code
The Couplers	Message Chain
	Middle Man
	Feature Envy
	Inappropriate Intimacy

Table 3.2. Mäntylä Taxonomy (2006) [53]

3.1.2. Inter- and Intra-Class Taxonomy

Getting to know the different propositions develops a strong intuition on distinguishing individual smells and ascribing to them some distinguishing features. Obstruction categorization is not the only one found in the literature. For example, a proposal from Atwood's blog page from 2006 [6] breaks down the Code Smells into only two categories:

— Code Smells Within Classes,
— Code Smells Between Classes.

This shows that the problems found from the code perspective can be considered from a narrow class spectrum, not going beyond the abstraction of a single object but also at the level of interclass communication and connections between classes in the project as a whole. This attribute appeared 2 months later in *"Extending a taxonomy of bad code smells with metrics"* as the `INTRA` boolean attribute [54], describing whether the smell could be observed from an individual observation without having available information on other related components. In 2010, Moha did a similar thing. He divided the smells and antipatterns according to their lexical, structural, and measurable properties [60].

Code Smell	**Jeff Atwood Grouping**	**Marticorena et al. 2006**
Comments	Within Classes	Inter Related
Long Method		Individual (Intra)
Long Parameter List		Individual (Intra)
Duplicated Code		Inter Related
Conditional Complexity		Individual (Intra)
Combinatorial Explosion		—
Large Class		Individual (Intra)
Type Embedded in Name		—
Uncommunicative Name		—
Inconsistent Names		—
Dead Code		—
Speculative Generality		Individual (Intra)
Oddball Solution		—
Temporary Field		Individual (Intra)
AC w/ DI	Between Classes	Inter Related
Primitive Obsession		Inter Related
Data Class		Individual (Intra)
Data Clumps		—
Refused Bequest		—
Inappropriate Intimacy		Inter Related
Indecent Exposure		—
Feature Envy		Inter Related
Lazy Class		Individual (Intra)
Message Chain		Inter Related
Middle Man		Inter Related
Divergent Change		Inter Related
Shotgun Surgery		Inter Related
Parallel Inheritance Hierarchies		Inter Related
Incomplete Library Class		Inter Related

Table 3.3. Atwood [6] and Marticorena Grouping [54]

3.1.3. Wake Taxonomy

In *"Refactoring Workbook"*, Wake created the most numerous groups of *Code Smells*. He listed nine groups, which I define as grouping by *occurrence* where *Code Smell* may appear:

— *Names*,
— *Conditional Logic*,
— *Message Calls*,
— *Unnecessary Complexity*,
— *Responsibility*,
— *Interfaces*,
— *Data*,
— *Duplication*,
— *Measured Smells*.

Additionally, he expands on the original Fowler list by adding new code smells regarding the naming (tokens, variables, methods, classes, functions) and the understandability of the processed code, such as conditional logic [92]. One could say that this is a more detailed version of Atwood's grouping, but it has more specifics: Instead of just deciding whether the scope of the issue is class-related or broader, it explicitly names the location of detection. Both of these classifications can be useful. I have called it the *occurrence* grouping because it answers the question, *"Where is the smelly code?"*: *"I saw that the problem lies in the conditional logic and data."*, *"The interfaces in these two files are smelly."*, *"The length of this method smells."*.

3.1.4. Clean Code Grouping

Speaking of code quality in general, I cannot forget to mention an important book, "Clean Code", written by Martin, in which another grouping appeared in the Smells and Heuristics Chapter [55]. The list is as follows: *Comments*, *Environment*, *Functions*, *General*, *Names*. In this list, the previously mentioned code smell *Comments* appear as an individual category (more on that in the Code Smells Discussion Chapter, "Issue with Comments" Section 5.1). In contrast, many other smells have received new names and similar explanations or definitions.

It is not easy to draw a general conclusion about the grouping of this list for two reasons. First, there is a "General" category, a "black box" of smells containing 36 of them. The *Comments*, *Environment*, and *Function*, have only 5, 2, and 4 smells, respectively, which is a visible disproportion. Having a *General* or *Other* type category is usually very encouraging to abuse because of its generality (rather than looking for a better fit in another category, distinctively named, the smell can be put in the *General* group). Second, there is not enough differentiation within the *General* category.

Group Name	**Code Smell**
Measured Smells	Comments
	Long Method
	Large Class
	Long Parameter List
Duplication	Duplication
	Alternative Classes with Different Interfaces
Data	Primitive Obsession
	Data Class
	Data Clump
Interfaces	Incomplete Library Class
	Refused Bequest
Responsibility	Feature Envy
	Inappropriate Intimacy
	Divergent Change
	Shotgun Surgery
	Parallel Inheritance Hierarchies
	Combinatorial Explosion
Unnecessary Complexity	Dead Code
	Lazy Class
	Speculative Generality
Message Calls	Message Chains
	Middle Man
Conditional Logic	Null Check
	Complicated Boolean Expressions
	Special Case
	Switch Statement
Names	Names with Embedded Types
	Uncommunicative Names
	Inconsistent Names

Table 3.4. Wake Taxonomy (2004) [92]

Nevertheless, Martin also made many new propositions for new code smells that were not present or omitted in other taxonomies or lists but are worth mentioning. *Environment* group (including *"Build Requires More Than One Step"* and *"Tests Require More Than One Step"*) nowadays is already treated as a separate hierarchy of smells - they are not about the production code itself, but the number of command process steps to take the desired action, like running tests or building up a project. However, these concepts can be extrapolated to situations in the code, for example, when a class requires setting up code before it can be initialized or after it has done its job, but a teardown code is required to shut it down successfully. However, I will elaborate further on this idea of specifying *Environment Smells* as a new hierarchy. I already explained a bit the concept of *Bad Smells Hierarchies* in the background Section 1.5.2 but it needs a little more attention.

3.2. HIERARCHIES OF SMELLS

In Martin's *"Clean Code"* and in Smith's *"Refactoring Fundamentals"*, in addition to the Code Smell list, there is also a separate list for *Testing Code Smells*. I consciously skip over them because I would like to focus primarily on the (production) code-related smells in this thesis. This lack of elaboration does not mean that these are minor issues, but they constitute another large category that could be dissected and addressed in a separate work. I define those as a **separate hierarchy of smells**, trimming the *code* wording and leaving it as *Test Smells*. Many different publications have already created and studied such separate hierarchies, which I have already mentioned in the first chapter, in the *Bad Smells* Section 1.5.2.

These publications differentiate even more types of smell that may be separate entities from *Code Smells* or consider concepts that intertwine, such as *Architectural Smells* [28] or *Design Smells* [7]. Looking at their definitions, *architectural smells* are "the set of principal design decisions that govern a system" and [87] are usually made by violating the design principles or by wrong settlements during the decision process on the internal qualities of the software [23]. The *Design Smells* are "structures in the design that indicate a violation of fundamental design principles and negatively impact design quality" [86]. By examining them closely, one can come to similar ideas and concepts that will emerge in each of these categories only with a difference of **perspective** from which one took the view.

There are also concepts such as *Code Review Smell* that address various issues during the code review process [18]. Although there is no explicit process that should be undertaken during code review, there is a set of standard best practices and rules that both open-source projects and companies are converging on that should be followed. This does not have much in common with Code Smells but might be somewhat related to *Community Smells*, another subject that reflects on "suboptimal organizational and socio-technical patterns in the organizational structure of the software community" [68].

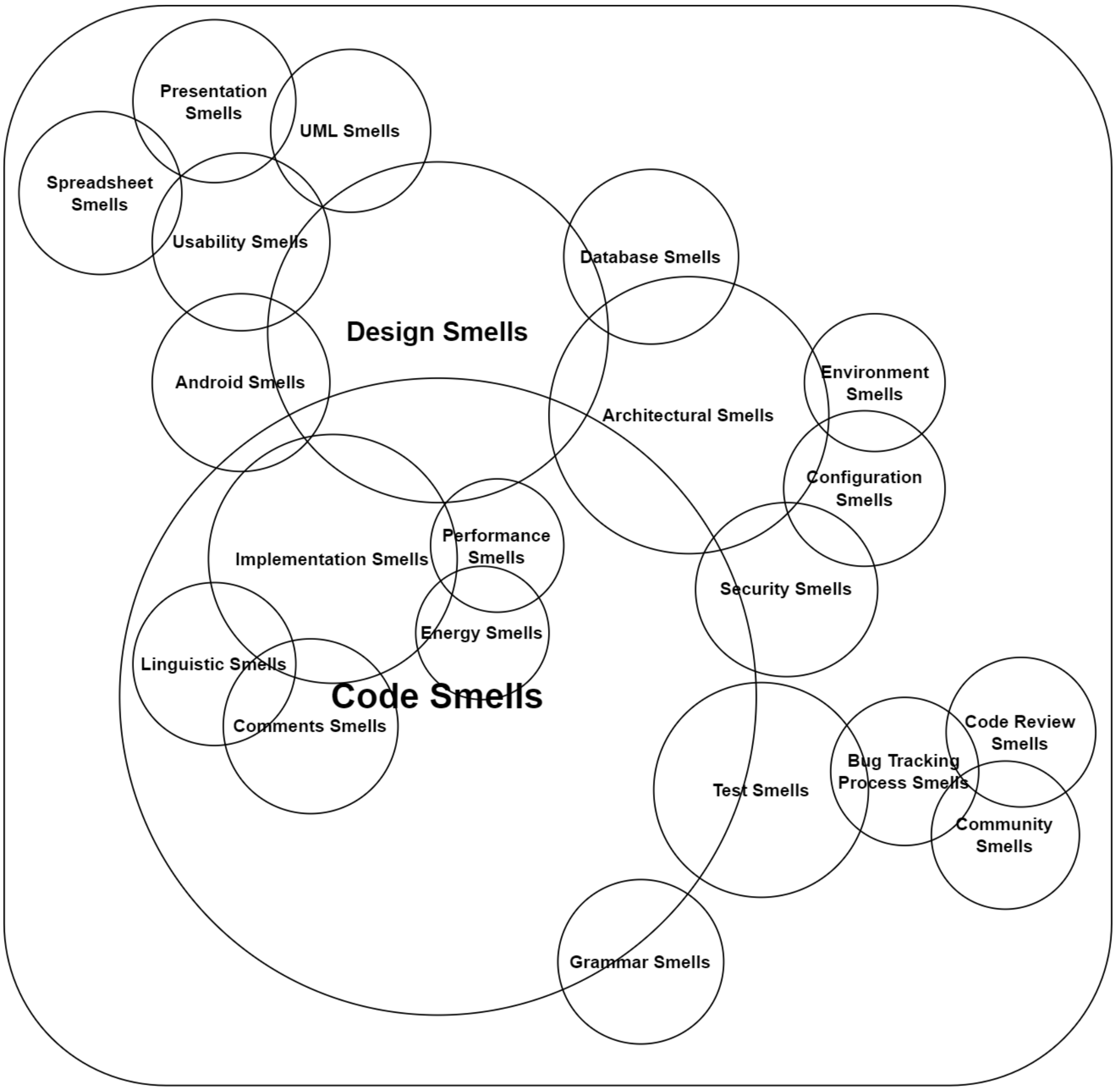


Fig. 3.1. Hierarchies of Bad Smells

The Vienna diagram in Figure 3.1 visualizes how hierarchies can intersect or differ from each other. For example, Comment Smells exist in the Code Smells hierarchy but, at the same time, constitute their individual hierarchy. At the same time, they have nothing to do with Performance Smells, but they have a common part with Linguistic Smells.

There are, in total, 22 different hierarchies, and it is not an easy task to find out about the existence of all of them. By moving only on the surface of bad smells, one can barely receive a good overview of the "original" code smells and maybe stumble upon the design smells. Having so many of them scattered around can be confusing and further decreases awareness of bad smell concepts. **There is a lack of a single source that would contain all of these smell hierarchies and explicitly mention them**. *Bad smell* is still used as an issue-defining prefix (*bad* code smell, *bad* design smell, *bad* linguistic smell, etc.), **but it could be a great umbrella term that refers to all of these smell hierarchies**.

3.3. VARIOUS AND SCATTERED USAGE OF TAXONOMIES

There is no one go-to source for all the up-to-date aggregated current code smell lists. There are numerous exciting studies summarizing the current literature, and each of them presents a different variation of the taxonomy, if any. Some latest examples:

— The simplified Mäntylä taxonomy (2006) classification in a tertiary systematic review from 2020 on code smells and refactorings [44] and the systematic review of the literature on bad smell machine-learning detection techniques [1]. The previous also mentions the categorization of *smells within*, *smells between*.

— The extended Mäntylä taxonomy (2003) in the 2020 systematic literature review article on the relationship between code smells and software quality attributes, [41] and the prioritization of code, smells review article from 2021 [42]. 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 [78].

— Only brief mentions of the existence of taxonomies as defined by Mäntylä, Wake, and Moha in a systematic review of the literature of 2019 on the detection of smells [72].

— Previously mentioned in Section 3.1.1 systematic review from 2019, in which the authors removed the *Change Preventers* and *Dispensables* while simultaneously introducing *Design Rule Abusers* and *Lexical Abusers* [72].

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, which I already mentioned in the background Section 1.5.1. Therefore, aggregating all information in an easily accessible form is attractive.

3.4. SUMMARY

Gathering all available information, I have found three significant groups that appeared in the literature by which *Code Smells* can be categorized.

1. **Obstruction Grouping**
   - Informing about the type of general problem they are causing.
   - This is the most common taxonomy for code smells.
   - Inspired by:
     - Mäntylä: *"A Taxonomy for „Bad Code Smells”"* (2003) [51],
     - Smith: *"Refactoring Fundamentals"* (2013) [83].
2. **Expanse Grouping**

    - Tells whether the *Code Smell* scope is narrow (within a class) or wide (between classes).
    - Inspired by:
        - Atwood: *"Code Smells"* (2006) [6],
        - Moha: *"Decor: A method for the specification and detection of code and design smells"* (2009) [60],
        - Marticorena: *"Extending a taxonomy of bad code smells with metrics"* (2013) [54].
3. **Occurrence Grouping**
    - This is the localization, where (or method by which) the *Code Smell* can be detected.
    - Inspired by:
        - *Wake: "Refactoring workbook" (2004) [92].*

I created a table that aggregates the scattered list of code smells and presents them in an easily digestible form - Table 3.5 and Table 3.6. The table covers all the code smells that I have found in the literature (both in the area of Code Smells and other sources of software engineering that did not strictly mention code smells) and in the gray knowledge area (discussion forums and various pages), which I listed in the Code Smell list chapter [ref. Chapter 4]. They are ordered chronologically with respect to the authors who first mentioned the smell.

Some of the names for the individual Code Smells have been updated to their latest form. Few of them were updated by the original author himself. For the rest, I faithfully changed the nomenclature following the original convention while removing any ambiguity or controversy that could arise. The name update list is as follows:

- "Switch Statement" -> "Repeated Switching" -> **"Conditional Complexity"**
    - Switch Statement was updated by Fowler himself in his newest book, *"Improving the Design of Existing Code - Second Edition"* (2018) [26].
    - Conditional Complexity term from *"Refactoring Fundamentals"* (2013) proposed by Smith seems to be the best fit for the name [83].
    - Repeated Switching could be in the more precise *Implementation Smells* hierarchy.
- "Lazy Class" -> **"Lazy Element"**
    - Updated by the Fowler himself in his newest book *"Improving the Design of Existing Code - Second Edition"* (2018) [26].
- "Inappropriate Intimacy" -> **"Insider Trading"**
    - Updated by the Fowler himself in his newest book *"Improving the Design of Existing Code - Second Edition"* (2018) [26].
- "Base Classes Depending on Their Derivatives" -> **"Base Class Depends on Subclass"**
    - Shorter alternative proposed by Smith in *"Refactoring Fundamentals"* (2013) [83].

Two of the smells currently present in the literature, which aroused the tremendous controversy with a conditional understanding of their terminology, were entirely removed from the Code Smell list. They received their successors. These are *Data Classes* and *Comments*. I elaborate further on this decision in the Code Smell Discussion chapter in *Issue with Comments* (Section 5.1) and *Issue with Data Classes* (Section 5.2) which is entirely devoted to explaining this decision.

Code Smell	**Obstruction**	**Expanse**	**Occurrence**
Fowler (2003)			
Long Method	Bloaters	Within	Measured Smells
Large Class	Bloaters	Within	Measured Smells
Long Parameter List	Bloaters	Within	Measured Smells
Primitive Obsession	Bloaters	Between	Data
Data Clumps	Bloaters	Between	Data
Temporary Fields	O-O Abusers	Within	Data
Conditional Complexity	O-O Abusers	Within	Conditional Logic
Refused Bequest	O-O Abusers	Between	Interfaces
AC with DI	O-O Abusers	Between	Duplication
Parallel Inh. Hierarchies	Change Prevent.	Between	Responsibility
Divergent Change	Change Prevent.	Between	Responsibility
Shotgun Surgery	Change Prevent.	Between	Responsibility
Lazy Element	Dispensables	Between	Unn. Complexity
Speculative Generality	Dispensables	Within	Unn. Complexity
Dead Code	Dispensables	Within	Unn. Complexity
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	Measured Smells
Incomplete Library Class	Other	Between	Interfaces
Wake (2004)			
Uncommunicative Name	Lexical Abusers	Within	Names
Magic Number	Lexical Abusers	Within	Names
Inconsistent Names	Lexical Abusers	Within	Names
Type Embedded In Name	Couplers	Within	Names
Combinatorial Explosion	Obfuscators	Within	Responsibility
Comp. Boolean Expressions	Obfuscators	Within	Conditional Logic
Conditional Complexity	O-O Abusers	Within	Conditional Logic
Null Check	Bloaters	Between	Conditional Logic

Table 3.5. Proposed Taxonomy as of 2022 *(part 1/2)*

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

Table 3.6. Proposed Taxonomy as of 2022 *(part 2/2)*

4. THE CURRENT LIST OF CODE SMELLS

This sections contains 56 Code Smells in total, of which 16 are new original propositions. These smells are listed in the tables mentioned in Chapter 3 on Code Smell Taxonomies (see Table 3.5 and Table 3.6). The content of this chapter exhaust the current available information on particular elements that can be classified as such, and which are not language specific. The list is in alphabetical order, divided into sections by the "classic" obstruction hierarchy.

4.1. BLOATERS

The first category of *Code Smells* — **Bloaters** — can be summarized by the sentence *"the code that just does not need to be that big"*. This refers to any method or classes that have an unproportionally large size to what they do, making them difficult to work with. Usually, they are not a problem right away, but rather, they start appearing over time, when new features are developed, and so — increasingly more code insertions are made to existing functions, which after enough time, make them smell.

4.1.1. Combinatorial Explosion

Origin: William C. Wake, "Refactoring Workbook" (2004) [92]
Obstruction: Bloater
Expanse: Within a Class
Occurrence: Responsibility
Hierarchies: Code Smell, Design Smell

The *Combinatorial Explosion* occurs when a lot of code does almost the same thing - here, the word "almost" is crucial. The number of cases needed to cover every possible path is massive, as is the number of methods. You can grasp a solid intuition of this smell by thinking about code blocks that differ from each other only by the quantities of data or objects used in them. Wake specifies that "(...) this is a relative of *Parallel Inheritance Hierarchies* Code Smell [Section 4.2.5], but everything has been folded into one hierarchy." [92].

Causation:

Instead, what should be an independent decision, gets implemented via a hierarchy. Let us suppose that someone organized the code so that it queries an API by a specific method with specific set-in conditions and data. Sooner or later, there are just so many of these methods as the need for different queries increases in demand.

Problems:

1. Don’t Repeat Yourself Principle Violation
   - Introducing new functionality requires multiple versions to be introduced in various places.
2. Open-Closed Principle Violation
   - Class is not closed for modification if it "prompts" the developer to add another `elif`.

```python
class Minion:
    name: str
    state: 'ready'

    def action(self):
        if self.state == 'ready':
            self.animate('standing')
        elif self.state == 'fighting':
            self.animate('fighting')
        elif self.state == 'resting':
            self.animate('resting')

    def next_state(self):
        if self.state == 'ready':
            return 'fighting'
        elif self.state == 'fighting':
            return 'resting'
        elif self.state == 'resting':
            return 'ready'

    def animate(self, animation: str):
        print(f"{self.name} is {animation}!")
```

Listing 4.1. Combinatorial Explosion Code Smell

Refactor Methods:

1. Replace Inheritance with Delegation,
2. Tease Apart Inheritance.

```python
class State(ABC):
    @abstractmethod
    def next() -> State:
        """ Return next State """

    @abstractmethod
    def animate() -> str:
        """ Returns a text-based animation """

class Ready(State):
    def next():
        return States.FIGHT

    def animate():
        return 'standing'

class Fight(State):
    def next():
        return States.REST

    def animate():
        return 'fighting'

...

class States(Enum):
    READY: State = Ready
    FIGHT: State = Fight
    REST: State = Rest

class Minion:
    name: str
    state: State = States.Ready

    def action(self):
        self.state.animate()

    def next_state(self):
        self.state = self.state.next()

    def animate(self, animation: str):
        print(f"{self.name} is {self.state.animate()}!")
```

Listing 4.2. Combinatorial Explosion Fix: Use State Pattern

4.1.2. Data Clumps

Origin: Fowler M., Kent B., "Refactoring: Improving the design of existing code" (1999) [56]
Obstruction: Bloater
Expanse: Between Classes
Occurrence: Data
Hierarchies: Code Smell, Design Smell

Data Clumps refer to a situation in which a few variables are passed around many times in the codebase instead of being packed into a separate object. Think of it as having to hold different groceries in a grocery store by hand instead of putting them into a basket or at least a handy cardboard box - this is just not convenient. Any set of data items that are permanently or almost always used together, but are not organized jointly, should be packed into a class. An example could be the `RGB` values held separately rather than in an `RGB` object.

Causation:
Developers often believe that a pair of variables is unworthy of creating a separate instance for them that could aggregate them under a common abstraction [92].

Problems:

1. Hidden Abstraction
   - Variables grouped into objects of their own increase the readability of the code, thus making the concept clearer.
2. More Complex APIs
   - Components Interfaces complexity increases with the number of accepted data.

```python
def colorize(red: int, green: int, blue: int):
    ...
```

Listing 4.3. Data Clumps Code Smell

Refactor Methods:

1. Extract Class,
2. Introduce Parameter Object.

```python
@dataclass(frozen=True)
class RGB:
    red: int
    green: int
    blue: int


def colorize(rgb: RGB):
    ...
```

Listing 4.4. Data Clumps Fix

4.1.3. Large Class

Known As: Blob, Brain Class, Complex Class, God Class, God Object, Schizophrenic Class, Ice Berg Class
Origin: Fowler M., Kent B., "Refactoring: Improving the design of existing code" (1999) [56]
Obstruction: Bloater
Expanse: Within a Class
Occurrence: Measured Smells
Hierarchies: Antipattern, Architecture Smell, Code Smell, Design Smell

When one combines the smell of *Long Method* [Section 4.1.4] and *Long Parameter List* [Section 4.1.5] but on a higher abstraction level, he would get the *Large Class* code smell. Many long-form methods and an abundant number of parameters that can be passed to a class cause *Large Class* problems. The point is that the class has too many responsibilities and is doing too much.

Causation:

This problem occurs because, under time constraints, it is much easier to place a new code in an existing class than to create a whole new class for the feature.

Problems:

1. Hard to Read and Understand
   - Large components are hard to comprehend, which amplifies if the component is not purely functional and stateless.
2. Hard to Change
   - It is tough to assess where the change has to be made, and even after it is done, the developer has to verify whether that was the only one in the class that had to be changed to implement his new desired functionality. There is also the risk of breaking the functionalities of all the other responsibilities that the class has and breaking by making unexpected side effects.
3. Hard to Test

- The larger the class, the more potential scenarios (all the methods and state variations) have to be covered via tests.

Refactor Methods:

1. Extract Class
2. Extract Subclass
3. Extract Interface
4. Extract Domain Object
5. Replace Data Value with Object

In the last three years, increased research in automatic Code Smell can be seen throughout the number of publications regarding the topic. These investigations required more specific terminology and precise definitions in addition to what Fowler wrote [56] in his book, which is why the vague Large Class can have more specific references:

> One thing we will not try to do here is to give you precise criteria for when a refactoring is overdue. In our experience, no set of metrics rivals informed human intuition. *Martin Fowler*

4.1.3.1. God Class

According to Lanza and Marinescu, *God Class* refers to classes that tend to centralize the system's intelligence [45]. The main characteristics are as follows:

1. It heavily accesses the data of other more straightforward classes (directly or using accessor methods).
2. It is large and complex.
3. There is a low cohesion between the methods belonging to that class (they perform a variety of different actions).
4. Rule of a Thumb: smell is said to occur when a class has 50 or more methods or attributes.

4.1.3.2. Brain Class

Brain Class is a complex class that accumulates intelligence like a brain. It is long and complicated. Has methods that centralize the brightness of a class [66].

4.1.3.3. Complex Class

A *complex Class* is a class with a complex control flow. It occurs when the cyclomatic complexity of a class is very high [86].

4.1.4. Long Method

Known As: Complex Method, God Method, Brain Method
Origin: Fowler M., Kent B., "Refactoring: Improving the design of existing code" (1999) [56]
Obstruction: Bloater
Expanse: Within a Class
Occurrence: Measured Smells
Hierarchies: Antipattern, Code Smell, Design Smell, Implementation Smell

One of the most apparent complications developers can encounter in the code is the length of a method. The more lines of code a function has, the more the developer has to strain himself mentally to comprehend what the particular block of code does thoroughly. The longer a procedure is, the more difficult it is to understand it [56]. It is also harder to change or extend [51]. In addition, reading more lines requires more time, which quickly adds up because the code is read more than it is written [55]. Fowler strongly believes in short methods as a better option.

Causation:

The author adds another code line rather than breaking the flow to identify the helper objects [92].

Problems:

1. Hard to Read and Reason About
   - Every time a developer wants to make any change to any part of it, he has to grasp the whole thing every time, which is time-consuming.
2. Low Reuse
   - Longer methods most likely have more functionalities, so developers cannot reuse them as easily as methods that are short and specific.
3. Side Effects
   - If a method is long, it is not very likely that it does only what the name of the method could indicate.

Refactor Methods:

1. Extract Method,
2. Replace Conditional with Polymorphism,
3. Replace Method with Command,
4. Introduce Parameter Object,
5. Preserve Whole Object,
6. Split Loop.

4.1.5. Long Parameter List

Origin: Fowler M., Kent B., "Refactoring: Improving the design of existing code" (1999) [56]
Obstruction: Bloater
Expanse: Within a Class
Occurrence: Measured Smells
Hierarchies: Antipattern, Code Smell, Design Smell, Implementation Smell

This is another code smell at the same abstraction level as *Long Method* [Section 4.1.4], which usually occurs when three, four, or more parameters are given as input for a single method. Basically, the longer the parameter list, the harder it is to understand [56].

Causation:

In an attempt to generalize a routine with multiple variations, a developer could have passed too many parameters at one point. Another causation could be due to ignorance of the object relationship between other objects, and thus, instead, calling in all the entities via parameters [92].

Problems:

1. Hard to Use
   - Usage of a method with many parameters requires more knowledge to use it.
   - They come inconsistently because they change every time there's a need for more data.
2. Increased Complexity
   - The input value is highly inconsistent, which creates too much variety in what might happen throughout the execution.
3. Single Responsibility Principle Violation
   - When there are too many parameters, most likely, the method tries to do too many things or has too many reasons to change.

```
def foo(author: str, commit_id: str, files: List[str], sha_id: str, time:
↪   str):
    ...

author, commit_id, files, sha_id, time = get_last_commit()
foo(author, commit_id, files, sha_id, time)
```

Listing 4.5. Long Parameter List Code Smell

```
@dataclass(frozen=True)
class Commit:
    author: str
    commit_id: str
    files: List[str]
    sha_id: str
    time: str

    def foo(self):
        ...

commit = Commit(**get_last_commit())
commit.foo()
```

Listing 4.6. Long Parameter List Fix

Refactor Methods:

1. Replace Parameter with Query,
2. Preserve Whole Object,
3. Introduce Parameter Object,
4. Remove Flag Argument,
5. Combine Methods into Class.

4.1.6. Null Check

Origin: William C. Wake, "Refactoring Workbook" (2004) [92]
Obstruction: Bloater
Expanse: Between Classes
Occurrence: Measured Smells
Hierarchies: Antipattern, Code Smell, Design Smell, Implementation Smell

Null check is widespread everywhere because the programming languages allow it. It causes a multitude of `undefined` or `null` checks everywhere: in guard checks, condition blocks, and verifications clauses. Instead, special objects could be created that implements the missing-event behavior, errors could be thrown and caught, and developers would remove thus many duplications. Even an anecdote sometimes appears here and there on discussion forums that the inventor of the `null` reference, Tony Hoare (also known as the creator of the QuickSort algorithm), apologizes for its invention and calls it a *billion-dollar mistake*.

Causation:

The direct cause of null checking is the lack of a proper Null Object that might implement the object's behavior in case it's null. There is a strong opinion that `null` or `undefined` is a detrimentally bad idea in programming languages [92].

Problems:

1. Duplication
   - Usually the null check is reoccurring.
2. Increased Complexity
   - Special case must be made for an object that might be undefined.
3. Bijection Violation
   - A `null/undefined` model is not in a one-to-one relationship with the domain. Moreover, there is no representation.

Refactor Methods:

1. Introduce Null Object,
2. Introduce `Maybe`,
3. Introduce `Optional`.

```
class BonusDamage(ABC):
    @abstractmethod
    def increase_damage(self, damage: float) -> float:
        """ Increases the output damage """


class Critical(BonusDamage):
    multiplier: float

    def increase_damage(self, damage: float):
        def additional_damage() -> float:
            return damage * self.multiplier * math.random(0, 2)

        return damage + additional_damage()


class Magical(BonusDamage):
    multiplier: float

    def increase_damage(self, damage: float):
        return damage * multiplier


bonus_damage: BonusDamage | None = perk.get_bonus_damage()

def example_of_doing_something_with_bonus_damage(bonus_damage: BonusDamage
↪   | None) -> ... | None:
    if not bonus_damage:
        return

    ...
```

Listing 4.7. Null Check Code Smell

```
class BonusDamage(ABC):
    @abstractmethod
    def increase_damage(self, damage: float) -> float:
        """ Increases the output damage """


class Critical(BonusDamage):
    multiplier: float

    def increase_damage(self, damage: float):
        def additional_damage() -> float:
            return damage * self.multiplier * math.random(0, 2)

        return damage + additional_damage()


class Magical(BonusDamage):
    multiplier: float
    def increase_damage(self, damage: float):
        return damage * multiplier


class NullBonusDamage(BonusDamage):
    def increase_damage(self, damage: float):
        return damage


bonus_damage: BonusDamage = perk.get_bonus_damage()

def example_of_doing_something_with_bonus_damage(bonus_damage:
↪   BonusDamage) -> ...:
    ...
```

Listing 4.8. Null Check Fix: Introduce Null Object

4.1.7. Primitive Obsession

Origin: Fowler M., Kent B., "Refactoring: Improving the design of existing code" (1999) [56]
Obstruction: Bloater
Expanse: Between Classes
Occurrence: Data
Hierarchies: Code Smell

Whenever a variable that is just a simple `string` or an `int` simulates a more abstract concept, which could be an object, we encounter a *Primitive Obsession* code smell. This lack of abstraction quickly becomes a problem whenever there is the need for any additional logic, and also because these variables easily spread wide and far in the codebase. This alleged verbal abstraction is just a "supposed" object, but it should have been a real object instead.

Causation:

Possibly a missing class to represent the concept in the first place. Mäntylä gives an example of representing money as primitive rather than creating a separate class [51], and so does Fowler, who states that many programmers are reluctant to develop their own fundamental types [26]. Higher-level abstraction knowledge is needed to clarify or simplify the code [43].

Problems:

1. Hidden Intention
   - The type does not correspond to the value that a variable holds (i.e. phone number as a `string`).
2. Lack of Encapsulation
   - These primitives often go in groups, and there's a lack of written relations between them and nothing that protects them from external manipulation.

```
birthday_date: str = "1998-03-04"
name_day_date: str = "2021-03-20"
```

Listing 4.9. Primitive Obsession Code Smell

Refactor Methods:

1. Replace Data Value with Object,
2. Extract Class,
3. Introduce Parameter Object,

```python
@dataclass(frozen=True)
class Date:
    year: int
    month: int
    day: int

    def __str__(self):
      return f"{self.year}-{self.month}-{self.day}"

birthday: Date = Date(1998, 03, 04)
name_day: Date = Date(2021, 03, 20)
```

Listing 4.10. Primitive Obsession Fix

4. Replace Array with Object,
5. Replace Type Code with Class,
6. Replace Type Code/Conditional Logic with State/Strategy,
7. Move Embellishment to Decorator.

4.1.8. Oddball Solution

Known As: Inconsistent Solution
Origin: Joshua Kerievsky, "Refactoring to Patterns" (2005) [43]
Obstruction: Bloater
Expanse: Between Classes
Occurrence: Duplication
Hierarchies: Code Smell

If a similar problem is solved differently in different parts of the project, it is an Oddball Solution. This code smell could also have been classified under *Duplicated Code* Code Smell [Section 4.4.2], although it is not exactly a one-to-one copy-paste - it is more subtle [43].

Causation:

This smell often occurs when there is some recognized method of calling a set of classes whose interfaces are not uniform.

Problems:

1. Increased Complexity
   - There should be one way to deal with the problem throughout the project. There is no reason to keep two ways of dealing with one of the problems - just use the better one.
2. Duplication

Refactor Methods:

1. Unify Interfaces with Adapter.

```python
class Instrument:
    ...

class USB2(Instrument):
    def __init__(self, ip, port):
        connection = socket.new_connection(f"{ip}:{port}")
        ...

    def ask(self, command):
        ...
        self.connection.query(command)


class USB3(Instrument):
    def __init__(self, address):
        connection = socket.new_connection(address)
        ...

    def read(self, command):
        ...
        self.connection.query(command)
```

Listing 4.11. Oddball Solution Code Smell

```
class SocketAdapter:
    def __init__(self, ip, port):
        connection = socket.new_connection(f"{ip}:{port}")
        ...

    def query(self, command):
        ...

class Instrument:
    connection: SocketAdapter
    ...

class USB2(Instrument):
    def __init__(self, ip, port):
        self.connection = SocketAdapter(ip, port)
        ...


class USB3(Instrument):
    def __init__(self, ip, port):
        self.connection = SocketAdapter(ip, port)
        ...
```

Listing 4.12. Oddball Solution Fix

4.1.9. Required Setup or Teardown Code

Known As: Inconsistent Solution
Origin: Steve Smith "Refactoring Fundamentals" (2013) [83]
Obstruction: Bloater
Expanse: Between Classes
Occurrence: Responsibility
Hierarchies: Code Smell

If, after the use of a class or method, several lines of code are required to:

- set it properly up,
- the environment requires specific actions beforehand or after its use,
- clean up actions are required,

then there is a *Required Setup or Teardown Code* code smell. Furthermore, this may indicate *Dubious Abstraction* [Section 4.2.3].

Causation:

Some functionality was taken beyond the class during development, and the need for their use within the class itself was overlooked.

Problems:

1. Lack of Cohesion
   - Class can't be reused by itself - it requires extra lines of code outside of its scope to make use of it.

Refactor Methods:

1. Replace Constructor with Factory Method,
2. Introduce Parameter Object.

```python
class Radio:
    def __init__(self, ip, port):
        socket = socket.connection(f"{ip}:{port}")

    ...

radio: Radio = Radio(ip, port)
...
# Doing something with the object
...
# Finalizing its use
radio.socket.shutdown(socket.shut_RDWR)
radio.socket.close()
...
```

Listing 4.13. Required Setup or Teardown Code Code Smell

```python
class Radio:
    def __init__(self, ip, port):
        socket = socket.connection(f"{ip}:{port}")

    def __del__(self):
        def graceful_shutdown():
            self.socket.shutdown(socket.shut_RDWR)
            self.socket.close()

        graceful_shutdown()
        super().__del__()

    ...

radio: Radio = Radio(ip, port)
...
# Doing something with the object
...
# Finalizing its use no longer requires manual socket closing
...
```

Listing 4.14. Required Setup or Teardown Code Fix

4.2. CHANGE PREVENTERS

The Change Preventers hinder the change or further development of the software [51]. They all violate the rule suggested by Fowler and Beck, which says that classes and possible changes should have bijection relationship (one-to-one) [56].

4.2.1. Callback Hell

Known As: Hierarchy of Callbacks, Pyramid of Doom
Origin: Marcel Jerzyk, "Code Smells: A Comprehensive Online Catalog and Taxonomy" (2022)
Obstruction: Change Preventers
Expanse: Within a Class
Occurrence: Conditional Logic
Hierarchies: Code Smell

Smell with a scent similar to the Conditional Complexity [Section 4.6.3] where tabs are intended deep and the 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. One of the most popular callbacks could be the `addEventListener` in JavaScript.

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

Refactor Methods:

1. Extract Methods,
2. Use Asynchronous Function,
3. Use Promises.

```
const makeSandwich = () => {
  ...
  getBread(function(bread) {
    ...
    sliceBread(bread, function(slicedBread) {
      ...
      getJam(function(jam) {
        ...
        brushBread(slicedBread, jam, function(smearedBread) {
          ...
        });
      });
    });
  });
};
```

Listing 4.15. Callback Hell Code Smell

```
const getBread = doNext => {
  ...
  doNext(bread);
};

const sliceBread = doNext => {
  ...
  doNext(breadSlice);
};

...

const makeSandwich = () => {
  return getBread()
    .then(bread => sliceBread(bread))
    .then(jam => getJam(beef))
    .then(slicedBread, jam => brushBread(slicedBread, jam));
};
```

Listing 4.16. Callback Hell Fix

4.2.2. Divergent Change

Origin: Fowler M., Kent B., "Refactoring: Improving the design of existing code" (1999) [56]
Obstruction: Change Preventers
Expanse: Within a Class
Occurrence: Responsibility
Hierarchies: Code Smell, Design Smell

If adding a simple feature makes the developer change many seemingly unrelated methods inside a class, that indicates the *Divergent Change* code smell. Simply put, the class has irrelevant methods in it [35]. For example, suppose that someone needs to modify class ‘A ‘ due to a change in the database, but then has to modify the same class ‘A ‘ due to a change in the calculation formula [51].

The difference between *Divergent Change* and *Shotgun Surgery* [Section 4.2.6] is that the Divergent Change addresses the issue within a class, while the Shotgun Surgery between classes.

Causation:

Over time, a class tries to do more and more things and has many responsibilities. The fact that the class already has implemented two or more different types of decisions (for example, finding an object and doing something with it [92]) was overlooked and left unrefactored.

Problems:

1. Single Responsibility Principle Violation
   - Class have too much responsibility.
2. Duplication

Refactor Methods:

1. Extract Superclass, Subclass or new Class,
2. Extract Function,
3. Move Function.

```python
class ReportModifier:
    def get_report(self, report_name):
        ...
        return report

    def modify_report(self, report, new_entry):
        ...
        return modified_report

    def run(self, report_name, new_entry):
        report = self.get_report(report_name)
        return self.modify_report(report, new_entry)


report_modifier = ReportModifier(...)
modified_report = report_modifier.run('raport.csv', 'Parsed')
```

Listing 4.17. Divergent Change Code Smell

```python
class ReportReader:
    def get_report(self, report_name):
        ...
        return report

class ReportModifier:
    def modify_report(self, report, new_entry):
        ...
        return modified_report

report_reader = ReportReader(...)
report = report_reader.get_report('raport.csv')

report_modifier = ReportModifier(...)
modified_report = report_modifier.modify_report(report, 'Parsed')
```

Listing 4.18. Divergent Change Fix

4.2.3. Dubious Abstraction

Known As: Inconsistent Abstraction Levels, Functions Should Descend Only One Level of Abstraction, Code at Wrong Level of Abstraction, Choose Names at the Appropriate Level of Abstraction
Origin: Marcel Jerzyk, "Code Smells: A Comprehensive Online Catalog and Taxonomy" (2022)
Obstruction: Change Preventers
Expanse: Within a Class
Occurrence: Responsibility
Hierarchies: Code Smell, Design Smell

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 describes the above sentences as three separate code smells: *Functions Should Descend Only One Level of Abstraction*, *Code at Wrong Level of Abstraction*, and *Choose Names at the Appropriate Level of Abstraction* [55]. He observed that people are having trouble with it and often mix abstraction levels. Steve Smith, in his course, uses the term "Inconsistent Abstraction Levels".

I 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, but it might be misinterpreted by just recalling the meaning through its title. I 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. I wish no one ever heard, "that is how it always has been, and so it must continue to be done that way".

This frequent wrong selection of abstractions is why I decided to rename it to *Dubious Abstraction,* directly addressing the potential causation of the smell - to think about the code that one 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 the `Instrument` querying this message? Or is it the *connection device* doing it? Maybe I should extract it? Is `percentFull` a method of `Stack` for sure?

Causation:

It is difficult to grasp whether the abstraction or naming the developer gives to various entities is proper for psychological reasons, like Tunnel Vision. A correct solution requires stepping out of the box to think whether the abstraction is accurate, and that requires continuous and conscious mental action to be undertaken.

Problems:

1. Extendibility
    - Code written "too literally" is closed on extendibility. Disregarding any abstractions on the implementation ideas makes it hard to introduce new features.
2. Comprehensibility
    - Creating context without specifying ungeneralized concepts might be hard to follow. Varying abstraction levels make it challenging to create a mental map of the code workflow.
3. Single Responsibility Principle Violation
    - Methods with more than one layer of abstraction are most likely doing more than one thing.

Refactor Methods:

1. Extract Superclass,
2. Extract Subclass,
3. Extract new Class.

```python
from abc import ABC  # Abstract Base Class

class Instrument(ABC):
    adapter: ConnectionAdapter

    def reset(self):
        self.write("*RST")

    def write(self, command):
        ...
```

Listing 4.19. Dubious Abstraction Code Smell

```python
from abc import ABC

class Instrument(ABC):
    adapter: ConnectionAdapter

    def reset(self):
        self.adapter.write("*RST")
```

Listing 4.20. Dubious Abstraction Fix: Adapter

4.2.4. Flag Argument

Known As: Boolean in Method Parameter
Origin: Robert Cecil Martin, "Clean Code: A Handbook of Agile Software Craftsmanship" (2008) [55]
Obstruction: Change Preventers
Expanse: Within a Class
Occurrence: Conditional Logic
Hierarchies: Code Smell

Martin Fowler defines *Flag Arguments* as a "kind of function argument that tells the function to carry out a different operation depending on its value." [25]. There are two reasons why this is smelly. First of all - it can be a candidate for the *Boolean Blindness* code smell [Section 4.8.1]. Fowler gives a great example with the `Concert` class and `book(customer: Customer, is_premium: bool)` method. While reading the code, without knowing much more context, one will be stopped by invocation of this method: `book(marcel, false)` - excuse me, but precisely what "`false`"? The situation is clear if the method is divided into two separate parts instead of using a flag argument. Then, calling a method that provides more meaning through a name like `regularBook(marcel)` is much better in terms of comprehensibility [Listing 4.22].

The second problem is that it might be a cocoon phase before it develops into a beautiful full-fledged Conditional Complexity [Section 4.6.3]. First, what you see is `is_premium: bool`. The second time you come by, it is already transformed to `ticket_type: str`, switching through different options based on the value and the smell of the Primitive Obsession [Section 4.1.7] on top.

Causation:

Dirty introduction of new features is the root causation - it might be as easy and tempting as adding an `else if` clause to a conditional-checking block. The developer felt that it was just a tiny difference and did not bother to create a separate method for its implementation.

Problems:

1. Hard to Read
   - In most cases, you will not know what `false` or what `true` is going on until you hover over the method.
2. Hard to Change
   - Boolean as a flag argument implies that a method has two ways of working.

Refactor Methods:

1. Remove Flag Argument,
2. Extract Method.

```
class Concert:
    def book(self, customer: Customer, is_premium: bool):
        if is_premium:
            ...
        else:
            ...

book(marcel, false) # ? false what
```

Listing 4.21. Flag Argument Code Smell

```
class Concert:
    def book_premium(self, customer: Customer):
        ...

    def book_regular(self, customer: Customer):
        ...

book_regular(marcel)
```

Listing 4.22. Flag Argument Fix: Extract Method

4.2.5. Parallel Inheritance Hierarchies

Origin: Fowler M., Kent B., "Refactoring: Improving the design of existing code" (1999) [56]
Obstruction: Change Preventers
Expanse: Between Classes
Occurrence: Responsibility
Hierarchies: Code Smell

This occurs when an inheritance tree depends on another inheritance tree by composition, and to create a subclass for a class, one finds that he has to make a subclass for another class. Fowler specified that this is a special case of *Shotgun Surgery* code smell [Section 4.2.6].

Causation:

This smell can happen naturally when trying to model a problem in a domain. The problem arises when these hierarchies are created artificially and unnecessarily (for example, by adding a standard prefix throughout the classes).

Problems:

1. Duplication
   - Requires additional work to be done, which might be redundant.

Refactor Methods:

1. Move Method,
2. Move Field,
3. Create Partial,
4. Fold Hierarchy into One.

```
class User(ABC):
    ...
    functions: Functions

class Functions(ABC):
    ...



class BasicUser(User):
    ...

class BasicFunctions(Functions):
    ...




class PremiumUser(User):
    ...

class PremiumFunctions(Functions):
    ...

# each time a new user is added, so is a new function subclass with the
↪   same prefix
```

Listing 4.23. Parallel Inheritance Hierarchies Code Smell

```python
class Animal(ABC):
    ...
    food: Food

class Food(ABC):
    ...



class Elephant(Animal):
    ...
    food: Vegan

class Vegan(Animal):
    ...




class Lion(Animal):
    ...
    food: Carnivore

class Carnivore(Food):
    ...

# domain-like mapping might eventually be reused
```

Listing 4.24. Parallel Inheritance Hierarchies Fix

4.2.6. Shotgun Surgery

Known As: Solution Sprawl
Origin: Fowler M., Kent B., "Refactoring: Improving the design of existing code" (1999) [56] as "Shotgun Surgery" and reappeared as "Solution Sprawl" in "Refactoring to Patterns" (2005) by Joshua Kerievsky [43]
Obstruction: Change Preventers
Expanse: Between Classes
Occurrence: Responsibility
Hierarchies: Code Smell

Similar to *Divergent Change* [Section 4.2.2], but with a broader spectrum, the smell symptom of the *Shotgun Surgery* code is detected by the unnecessary requirement of changing multiple different classes to introduce a single modification. Things like that can happen with the failure to use the correct design pattern for the given system. This expansion of functionality can lead to an easy miss (and thus introduce a bug) if these small changes are all over the place and they are hard to find. Most likely, too many classes solve a simple problem [Section 4.1.8].

Karievsky noted this smell as *Solution Sprawl* [43]. Monteiro stated that the tiny difference between these two comes from how they are sensed. In the *Divergent Change*, one becomes aware of the smell while making the changes, and in *Solution Sprawl*, one is mindful by observing the issue [62].

Causation:

Wake says it could have happened due to an "overzealous attempt to eliminate *Divergent Change*" [92]. A missing class could understand the entire responsibility and handle the existing cluster of changes by itself. That scenario could also happen with cascading relationships of classes [35].

Problems:

1. Single Responsibility Principle Violation
   - The codebase is non-cohesive.
2. Duplicated code
   - The increased learning curve for new developers to effectively implement a change.

Refactor Methods:

1. Extract Method,
2. Combine Functions into Class,
3. Combine Functions into Transform,
4. Split Phase,
5. Move Method and Move Field,
6. Inline Function/Class.

```python
class Minion:
    energy: int

    def attack(self):
        if self.energy < 20:
            animate('no-energy')
            skip_turn()
            return
        ...

    def cast_spell(self):
        if self.energy < 50:
            animate('no-energy')
            skip_turn()
            return
        ...

    def block(self):
        if self.energy < 10:
            animate('no-energy')
            skip_turn()
            return
        ...

    def move(self):
        if self.energy < 35:
            animate('no-energy')
            skip_turn()
            return
        ...
```

Listing 4.25. Shotgun Surgery Code Smell

```python
class Minion:
    energy: int

    def attack(self):
        if not self.has_energy(20):
            return
        ...

    def cast_spell(self):
        if not self.has_energy(50):
            return
        ...

    def block(self):
        if not self.has_energy(10):
            return
        ...

    def move(self):
        if not self.has_energy(35):
            return
        ...

    def has_energy(self, energy_required: int) -> bool:
        if self.energy < energy_required:
            self.handle_no_energy()
            return False
        return True

    def handle_no_energy(self) -> None:
        animate('no-energy')
        skip_turn()
```

Listing 4.26. Shotgun Surgery Fix

4.2.7. Special Case

Known As: Complex Conditional
Origin: William C. Wake, "Refactoring Workbook" (2004) [92]
Obstruction: Change Preventers
Expanse: Within a Class
Occurrence: Conditional Logic
Hierarchies: Code Smell, Implementation Smell

Wake addresses the complex conditional situation as a *Special Case* code smell with two symptoms - a complex `if` statement or value checking before doing the actual work [Section 4.1.9] [92].

Causation:

There was a need for a special case to handle. A hotfix that was never adequately fixed could also be the reason for the smell.

Problems:

1. Comprehensibility
   - The method is doing a specific task, but there is "one special case" to consider.
2. Increased Test Complexity
   - Special case has to have an extra special test.

```
def parse_foo(foo: Foo) -> Goo:
    if 'zoo' in foo.sample_attribute:
        ...
        ...

    ...
```

Listing 4.27. Special Case Code Smell

Refactor Methods:

1. Consolidate Conditional Expression,
2. Replace Conditional with Polymorphism,
3. Introduce Null Object,
4. Replace Exception with Test.

Exceptions:

Recursive methods always have to have a base case to stop the recursion.

4.3. COUPLERS

The smells in the *Couplers* category indicate excessive coupling, which is against object-oriented design. This group could be a subcategory in *Object-Oriented Abusers*, although the smells listed here focus strictly on high coupling issues.

4.3.1. Afraid To Fail

Origin: Marcel Jerzyk, "Code Smells: A Comprehensive Online Catalog and Taxonomy" (2022)
Obstruction: The Couplers
Expanse: Within a Class
Occurrence: Responsibility
Hierarchies: Code Smell, Implementation Smell

The *Afraid To Fail* is a Code Smell name inspired by the common fear (at least among students [15]) of failure, which is professionally called *Atychiphobia* [71] - being scared of failure.

I am referencing it here because the fear of admitting failure (that something went wrong) is a common, relatable, psychological trait, and facing that fear would benefit 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, this behavior will clutter the code because, after a method or function call, additional code is required to check whether some status code is valid, a Boolean flag is marked, or the returned value is not `None`. And all of that is outside of the method scope.

If it is expected that a method might fail, then it should fail, either by throwing an `Exception` or, if not - it should return a special case `None`/`Null` type object of the desired class (following the *Null Object Pattern*), not null itself. For example, if an expected object cannot be received or created, and instead of some status indicator is sent back (which has to be checked after the method is completed), the smells it generates would be Afraid to Fail [Section 4.3.1] as well as Teardown Code [Section 4.1.9] code smell. Instead, following the *Fail Fast Principle*, the code should throw an error.

Problems:

1. Code Pollution
   - Requires additional `if` checks for return status or `null` checks.
2. Coupling
   - Creates artificial coupling with the method caller.
3. Fail Fast Violation
   - In system design, the Fail Fast concept is about reporting immediately when any condition is likely to indicate failure.

Refactor Methods:

1. Move Method.

```
def create_foo() -> dict:
    response = requests.get('https://api.github.com/events')
    if response.status_code != requests.codes.ok:
        return {'status_code': response.status_code, 'foo': None}
    return {
        'status_code': response.status_code,
        'foo': Foo(**response.json())
    }

foo_response: dict = create_foo()
if foo['status_code'] != requests.codes.ok:
    foo: Foo = foo_response['foo']
...
```

Listing 4.28. Afraid To Fail Code Smell

```
def create_foo() -> Foo:
    response = requests.get('https://api.github.com/events')
    if response.status_code != requests.codes.ok:
        raise Exception('Something went wrong')
    return Foo(**response.json())

foo: Foo = create_foo()
...
```

Listing 4.29. Afraid to Fail Fix: Raising Exception Immediately

4.3.2. Binary Operator In Name

Origin: Marcel Jerzyk, "Code Smells: A Comprehensive Online Catalog and Taxonomy" (2022)
Obstruction: The Couplers
Expanse: Within a Class
Occurrence: Responsibility
Hierarchies: Code Smell, Implementation Smell

This is straightforward: method or function names that have binary bitwise operators like `AND` and `OR` are obvious candidates for undisguised violators of the *Single Responsibility Principle* right out there in the open. If the method name has `and` in its name and then it does two different things, then one might ask why it is not split in half to do these two other things separately? Moreover, if the method name has `or`, then it not only does two different things but also, most likely, it has a stinky Flag Argument [Section 4.2.4], which is yet another code smell.

This code smell 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 the variables.

Problems:

1. Single Responsibility Principle Violation
   - Names with conjunction words strongly suggest that something is not responsible for just one thing.

Refactor Methods:

1. Extract Method.

```python
def render_and_save():
    # render logic
    ...
    # save logic
    ...
```

Listing 4.30. Binary Operator In Name Code Smell

```python
def render():
    # render logic
    ...

def save():
    # save logic
    ...
```

Listing 4.31. Binary Operator In Name Fix: Separate Methods

4.3.3. Fate over Action

Known As: Data Class
Origin: Marcel Jerzyk, "Code Smells: A Comprehensive Online Catalog and Taxonomy" (2022)
Obstruction: The Couplers
Expanse: Between Classes
Occurrence: Responsibility
Hierarchies: Code Smell, Design Smell

In Object-Oriented Programming, classes and their data go hand in hand with behavior. If a class has only the first part, it is an indicator that there could be a functionality tied to these variables, but it is missing or placed somewhere else.

Back in the days *(at time of defining - Fowler 2003) Data Class* would be enough to be classified as a bad smell, as it is the main evil of hidden and hard-to-debug problems in large-scale systems because of mutability - data were unexpectedly different, or it was missing at the point of execution due to some other unexpected reasons. Thus, the data were supposed to be closely tied to the logic to go hand in hand. This is also one of the main reasons functional programming is rising in popularity; one of its main principles is that the data should be immutable, so there would be no more of these sorts of bugs.

Data Class is still a sufficiently intuitive motive to follow. However, in programming languages, we have things like `Interface`-s or `Struct`-s to pack and type together a bunch of variables. This directly addresses and solves the *Data Clumps* code smell [Section 4.1.2]. Data Transfer Objects (DTO) are not uncommon with the dominance of web-based applications and communications over API. All of this could fall into the *Data Class* code smell category, but I rather believe we are not intentionally making everything stinky as programmers.

Data classes that cannot be changed (thus lacking setters, or with some sort of "frozen" property), are much less error-prone and should not be discouraged if they are a suitable fit to remove other smells or to pass data around, especially if the alternative is to have long un-verified dictionaries straight out of configuration file or API call. The immutable data class can have the additional benefit of verifying the types *(depending on a language)*, so if it is expected to have `address` given as a `string`, then that is a good Fail Fast mechanism (check Afraid to Fail [Section 4.3.1]) if instead a `None` or `undefined` is given due to a misformed or incomplete input. And after all, when the class is already here, it can be encouraging to fill it with further validation and behavior.

Causation:

It is common for classes to begin like this: you realize that some data are part of an independent object, so you extract it. But objects are about the commonality of behavior, and these objects are not developed enough to have much behavior yet. Wake (2004) [92]

Problems:

1. Tell, Don't Ask Principle Violation
   - The principle says that one should not ask about the object state to decide on action but rather straightforwardly send a command.
2. Coupling
   - Objects are unnecessarily coupled with each other when a particular class could take care of itself.

Refactor Methods:

1. Move Method,
2. Extract method,
3. Freeze Variables.

```python
@dataclass
class CommitManager:
    def update_author(commit: Commit, new_author: str):
        ...

    def update_message(commit: Commit, new_message: str):
        ...


@dataclass
class Commit:
    _author: str
    _message: str


commit_manager = CommitManager()
commit = Commit(
    author="Marceli Jerzyk",
    "Fix: Button Component styled width w/ rem (from px)"
)
commit_manager.update_author(commit, "Marcel Jerzyk")
```

Listing 4.32. Fate over Action Code Smell

```python
@dataclass
class Commit:
    _author: str
    _message: str

    def set_author(self, new_author: str):
        ...

    def set_message(self, new_message: str):
        ...


commit = Commit(
    author="Marceli Jerzyk",
    "Fix: Button Component styled width w/ rem (from px)"
)
commit.set_author(commit, "Marcel Jerzyk")
```

Listing 4.33. Fate over Action Fix: Move Methods

4.3.4. Feature Envy

Origin: Fowler M., Kent B., "Refactoring: Improving the design of existing code" (1999) [56]
Obstruction: The Couplers
Expanse: Between Classes
Occurrence: Responsibility
Hierarchies: Code Smell, Design Smell

If a method inside a class manipulates more features (be it fields or methods) of another class more than from its own, then this method has a *Feature Envy*. In Object-Oriented Programming, developers should tie the functionality and behavior close to the data it uses. The instance of this smell indicates that the method is in the wrong place and is more tightly coupled to the other class than to the one where it is currently located [51].

This was the explanation based on Fowler's book from 1999. In his recent "book update", he rephrased the *class* into *module*, generalizing the concept from a *zone* perspective. Depending on the size of the system, the *Feature Envy* code smell may apply accordingly.

Causation:

The root cause of this smell is misplaced responsibility.

Problems:

1. Low Testability
   - Difficult to create proper test or tests in separation. Mocking is required.

2. Inability to Reuse
   - Coupled objects have to be used together. This can cause lousy duplication issues if one tries to reuse applicable code by extracting and cutting off what he does not need.
3. Bijection Problems

Refactor Methods:

1. Move Method,
2. Move Field,
3. Extract Method.

```python
@dataclass(frozen=True)
class ShoppingItem:
    name: str
    price: float
    tax: float


class Order:
    ...
    def get_bill_total(items: list[ShoppingItem]) -> float:
        return sum([item.price * item.tax for item in items])

    def get_receipt_string(items: list[ShoppingItem]) -> list[str]:
        return [f"{self.name}: {self.price * self.tax}$" for item in
        ↪  items]

    def create_receipt(items: list[ShoppingItem]) -> float:
        bill = self.get_bill_total(items)
        receipt = self.get_receipt_string(items).join('\n')
        return f"{receipt}\nBill {bill}"
```

Listing 4.34. Feature Envy Code Smell

```python
@dataclass(frozen=True)
class ShoppingItem:
    name: str
    price: float
    tax: float

    @property
    def taxed_price(self) -> float:
        return self.price * self.tax

    def get_receipt_string(self) -> str:
        return f"{self.name}: {self.price * self.tax}$"

class Order:
    ...
    def get_bill_total(items: list[ShoppingItem]) -> float:
        return sum([item.taxed_price for item in items])

    def get_receipt_string(items: list[ShoppingItem]) -> list[str]:
        return [item.get_receipt_string() for item in items]

    def create_receipt(items: list[ShoppingItem]) -> float:
        bill = self.get_bill_total(items)
        receipt = self.get_receipt_string(items).join('\n')
        return f"{receipt}\nBill: {bill}$"
```

Listing 4.35. Feature Envy Fix: Moved Responsibility

4.3.5. Indecent Exposure

Known As: Excessive Exposure
Origin: Joshua Kerievsky, "Refactoring to Patterns" (2005) [43]
Obstruction: The Couplers
Expanse: Within a Class
Occurrence: Data
Hierarchies: Code Smell

Unnecessarily exposing internal details is an *Indecent Exposure* code smell. The methods and variables of a class that works only with other same class methods should be kept private.

Otherwise, it might lead to *Insider Trading* [Section 4.5.3] or *Feature Envy* [Section 4.3.4] code smells. One should always strive to hide as many variables and methods from other classes as possible. Exposing irrelevant code contributes to the complexity of a design.

Causation:

The developer could have a habit of creating all the methods public at first but then forgets to change the access levels to appropriate ones.

Problems:

1. Error Prone
   - Fields accessible from outside the class baits for unnecessary coupling issues.
2. Information Overload
   - There is no need to expose all information to everyone.

Refactor Methods:

1. Choose Proper Access Control,
2. Encapsulate Field,
3. Encapsulate Collection,
4. Hide Behind Method,
5. Hide Behind Abstract Class or Interface.

4.3.6. Type Embedded in Name

Origin: William C. Wake, "Refactoring Workbook" (2004) [92]
Obstruction: Lexical Abusers
Expanse: Within a Class
Occurrence: Names
Hierarchies: Code Smell, Implementation Smell, Linguistic Smell

```python
@dataclass
class Counter:
    count: int = field(init=False)

    def bump(self) -> None:
        self.count += 1

counter = Counter()
counter.bump()
print(f"Count: {counter.count}")
```

Listing 4.36. Indecent Exposure Code Smell

```python
@dataclass
class Counter:
    __count: int = field(init=False)

    @property
    def count() -> int:
        return self.__count

    def bump(self) -> None:
        self.__count += 1

    def reset(self) -> None:
        self.__count = 0

counter = Counter()
counter.bump()
print(f"Count: {counter.count}")
```

Listing 4.37. Indecent Exposure Fix

Whenever a variable has an explicit type prefix or suffix, it can strongly signal that it should be just a class of its own. For example, `current_date = "2021-14-11"`, embeds the potential class `Date` in the name of a variable, and the whole can be classified as the *Primitive Obsession* code smell [Section 4.1.7].

Wake signals that the embedded type could also be in the method names, giving an example of a `schedule.add_course(course)` function in contrast to `schedule.add(course)`. He notes that it could have been a matter of preference, although he insists that this can be a problem wherever some generalization occurs [92]. When a parent class for `Course` is introduced to cover not only *courses* but also *series of courses*, then `add_course()` has a name that is no longer appropriate, suggesting the usage of more neutral terms [92]. Parameters of a method are part of the method name, and this kind of naming is also a duplication.

With all the possibilities of annotating variables, it is unnecessary to mention it twice through variable name when type annotation or type hinting is present. Names with embedded types in them, for which no class yet could represent them, are good candidates to be refactored into separate classes.

Causation:

This was a standard in pointer-based languages, but it is not helpful in modern Object-Oriented Programming languages [92].

Problems:

1. Duplication
   - Both the argument and name mentions the same type.
2. Comprehensibility Issues
   - If the name of a variable is just precisely the name of the class, it's a case of *Uncommunicative Name* [Section 4.8.6].

Refactor Methods:

1. Extract Class,
2. Rename Method,
3. Rename Variable.

```
player_name: str = "Luzkan"
player_health: int = 100
player_stamina: int = 50
player_attack: int = 7

datetime = datetime.datetime.now()
foo()
datetime_2 = datetime.datetime.now()
```

Listing 4.38. Type Embedded In Name Code Smell

```
@dataclass
class Player:
    stamina: int
    health: int
    attack: int
    name: str

luzkan = Player(
    name="Luzkan"
    health=100,
    stamina=50,
    attack=7,
)

foo_bench_start_time = datetime.datetime.now()
foo()
foo_bench_end_time = datetime.datetime.now()
```

Listing 4.39. Type Embedded In Name Fix

4.4. DISPENSABLES

The **Dispensables** category includes some source of content that is probably unnecessary and should be removed [56]. The "content" word is not accidental because, in this category, there are also *Comments*, which are not present in the original taxonomy presented by Mäntylä (he listed them in the "Other" category [51]). Due to the aversion to placeholder types like "General" and "Other", which does not convey much meaning, *Dispensables* is a good place to place the original *Comments* Code Smell in. Besides the *Comments*, here are the *Data Class* and *Lazy Element* about which Fowler and Beck say that "if it is not doing enough, it needs to be removed or the responsibility has needs to be increased" [56], with the reason behind it that each element requires effort to understand and maintain. The *Speculative Generality*, *Duplicate Code*, and *Dead Code* concern the same issue, but in this case, it is probably just redundant or not used at all and should therefore be removed instead [51].

4.4.1. Dead Code

Origin: William C. Wake, "Refactoring Workbook" (2004) [92]
Obstruction: Dispensable
Expanse: Between Classes
Occurrence: Unnecessary Complexity
Hierarchies: Code Smell

If part of the code is not executed, it is a *Dead Code*. This code smell includes any place that the code executor will never reach, including the code that was commented out. Any `if` or `case` condition that cannot be reached, any code in methods and functions after the final `return` statement, or any code inside a `try except/catch` block that will never throw an error [55]. This smell usually is hard to detect and requires tool assistance [92].

Causation:

Never refactored long else-if blocks [Section 4.6.3] eventually have so many paths that no one even remembers if they are accessed anymore. Perhaps developers introduced a new way of working, and the old code was never cleaned up.

Problems:

1. You Ain't Gonna Need It Principle Violation
   - Old code that was not deleted but commented out *"just-in-case"* is just bloating the codebase.

Refactor Methods:

1. Remove It.

4.4.2. Duplicated Code

Known As: Clones, Code Clone, Duplicate Code, Common Methods in Sibling Class, External Duplication
Origin: Fowler M., Kent B., "Refactoring: Improving the design of existing code" (1999) [56]
Obstruction: Dispensable
Expanse: Within a Class
Occurrence: Duplication
Hierarchies: Code Smell, Design Smell, Implementation Smell

Duplicated Code does not need further explanation. According to Fowler, redundant code is one of the worst smells [56]. One thing is that this makes it more challenging to read the program. Checking whether the copies are identical is yet another issue - looking at whether there are no tiny differences between code blocks in search of Oddball Solution [Section 4.1.8] further unnecessarily absorbs developer time. Yet another thing is that whenever a change is made, one needs to check if this should have happened to just one or all of the existing copies of code, wherever they are.

Refactor Methods:

1. Extract Class,
2. Extract Function,
3. Pull Up Method,
4. Slide Statement,
5. Form Template Method.

4.4.3. Lazy Element

Known As: Lazy Class
Origin: Fowler M., Kent B., "Refactoring: Improving the design of existing code" (1999) [56] as *"Lazy Class"* renamed to "Lazy Element" in "Refactoring: Improving the Design of Existing Code" (2018) [26]
Obstruction: Dispensable
Expanse: Between Classes
Occurrence: Unnecessary Complexity
Hierarchies: Code Smell, Design Smell, Implementation Smell

A *Lazy Element* is an element that does not do enough. If a method, variable, or class does not do enough to pay for itself (for increased complexity of the project), it should be combined into another entity.

Causation:

This smell can happen due to the aggressive refactorization process in which the functionality of a class was truncated. It can also occur due to the unnecessary pre-planning - Speculative Generality [Section 4.4.4] - developer made it in anticipation of a future need that never eventuates.

Problems:

1. Increased Complexity
    - Projects complexity increases with the number of entities.

Refactor Methods:

1. Inline Class,
2. Inline Function,
3. Collapse Hierarchy.

```
class Strength:
    value: int

class Person:
    health: int
    intelligence: int
    strength: Strength
```

Listing 4.40. Lazy Element Code Smell

```
class Person:
    health: int
    intelligence: int
    strength: int
```

Listing 4.41. Lazy Element Fix

4.4.4. Speculative Generality

Origin: Fowler M., Kent B., "Refactoring: Improving the design of existing code" (1999) [56]
Obstruction: Dispensable
Expanse: Between Classes
Occurrence: Unnecessary Complexity
Hierarchies: Antipattern, Code Smell, Design Smell

Developers are humans, and humans are bad guessers [9]. Developers tend to create additional features in preparation for the future, guessing they will be helpful, but that time never came. This problem lies within the human psychological nature and, contrary to their best intentions, it just clutters the code.

Causation:

Psychologically, humans tend to anticipate scenarios and prepare for them.

Problems:

1. You Ain't Gonna Need It Principle Violation
   - The whole system is trying or expecting to do more than it is supposed to.
2. Increased Complexity
   - Each additional method, class, or module increases the required time and effort to understand it as a whole.

Refactor Methods:

1. Collapse Hierarchy,
2. Inline Function,
3. Inline Class,
4. Rename Method.

```
class Animal:
    health: int

class Human(Animal):
    name: str
    attack: int
    defense: int

class Swordsman(Human):
    ...

class Archer(Human):
    ...

class Pikeman(Human):
    ...
```

Listing 4.42. Speculative Generality Code Smell

```
class Human:
    name: str
    health: int
    attack: int
    defense: int

class Swordsman(Human):
    ...

class Archer(Human):
    ...

class Pikeman(Human):
    ...
```

Listing 4.43. Speculative Generality Fix

4.4.5. "What" Comment

Origin: Marcel Jerzyk, "Code Smells: A Comprehensive Online Catalog and Taxonomy" (2022)
Obstruction: Dispensables
Expanse: Within a Class
Occurrence: Names
Hierarchies: Code Smell, Implementation Smell

Recognizing all comments as Code Smells is controversial and raises many different opinions. For this reason, I 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* in a particular code section is happening, it is probably trying to mask some other Code Smell.

This distinction leaves room for the *"Why" Comments* which were already defined by Wake in 2004 and were considered helpful. Wake also notes that comments that cite non-obvious algorithms are also acceptable [92]. I 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 I have mentioned, the problem is that *Comments* are generally smelly. This is because they are a deodorant for other smells [56]. They may also quickly degrade with time and become another category of comments, *Fallacious Comments* [Section 4.8.2], which are a rotten, misleading subcategory of *"What" Comments*.

Causation:

The author sees that the code is confusing and tries to be helpful by adding explanations.

Problems:

1. Duplication
   - Bad docstrings, which are just duplicating everything that can be understood from the method signature (name, type and parameters) are cluttering the code, possibly sooner or later on the first occasion, they might become *Fallacious Comments*.
2. Cover up for other smells
   - A comment that must explain what is happening in the code indicates that it can't speak for itself, which is a strong indicator of present code smells.

Refactor Methods:

1. Extract Method,
2. Rename Method,
3. Introduce Assertion.

```python
class Foo:
    def goo(...):
        ...

        # Creating Report
        vanilla_report = get_vanilla_report(...)
        tweaked_report = tweaking_report(vanilla_report)
        final_report = format_report(tweaked_report)

        # Sending Report
        send_report_to_headquarters_via_email(final_report)
        send_report_to_developers_via_chat(final_report)
        ...
```

Listing 4.44. "What" Comment Code Smell

```python
class Foo:
    def goo(...):
        ...

        report = self.create_report(...)
        self.send_report(report)


    def create_report(self, ...):
        vanilla_report = get_vanilla_report(...)
        tweaked_report = tweaking_report(vanilla_report)
        return format_report(tweaked_report)

    def send_report(self, report):
        send_report_to_headquarters_via_email(final_report)
        send_report_to_developers_via_chat(final_report)
        ...
```

Listing 4.45. "What" Comment Fix

4.5. DATA DEALERS

Data Dealers is a new subcategory in the existing obstruction-like taxonomy created by Mäntylä. The smells within this category are all about potential problems that might arise due to the inevitable exchange of information within the system. Any implicit or explicitly undesirable behavior of data exchange falls into this category.

Within this category, two smells were previously classified as *Encapsulators* - *Message Chain* and *Middle Man*. Mäntylä extracted the *Encapsulators* category to underline their antagonistic behavior - decreasing one of them usually caused the other to increase to some degree [51]. Still, it could vary depending on the refactorization technique and the new proposed class hierarchy. However, the whole *Data Dealers* category retains this individual attribute for each smell. The puzzle of good design is to design such software to minimize these smells and balance out what must remain.

4.5.1. Global Data

Origin: Fowler M., Kent B., "Refactoring: Improving the Design of Existing Code" (2018) [26]
Obstruction: Data Dealers
Expanse: Between Classes
Occurrence: Data
Hierarchies: Code Smell, Design Smell

The *Global Data* is quite similar to the Middle Man code smell [Section 4.5.5], but here rather than a class, the broker is the *global scope* in which the data is freely available to everyone. These global scope variables are undesirable because it directly causes the Hidden Dependencies code smell [Section 4.5.2] and a highly nasty Mutable Data [Section 4.7.2] code smell. Global data can be read from anywhere, and there is no easy way to discover which bit of code touches it. If the variable in the global scope is additionally mutable, then this becomes an extremally bad case of Mutable Data, Fate over Action [Section 4.3.3] (data class). Fowler, in 1999 recalled that in the early days of programming, back when there were no objects, even causing the [Section 4.1.5] code smell was preferable to *Global Data* [56]. For the same reasons, a Singleton Pattern can also be problematic [26].

Problems:

1. Hard to Test
   - Each Global Variable is a hidden dependency that a tester has to mock.
2. Hard to Reason About
   - Global Variables exponentially fuzz the clarity of the codebase.
3. Inter Component Coupling
4. Encapsulation Violation

Refactor Methods:

1. Encapsulate Variable.

Exceptions:

In the context of the system as a whole, some communication between modules must take place. All possibilities should be appropriately balanced so that none of the smells dominate *Global Data* [Section 4.5.1], *Tramp Data* [Section 4.5.6], *Message Chain* [Section 4.5.4], *Middle Man* [Section 4.5.5], to make the entire codebase as clear as possible.

4.5.2. Hidden Dependencies

Origin: Steve Smith "Refactoring Fundamentals" (2013) [83], previously mentioned as Antipattern by Yu and Rajlich in "Hidden Dependencies in Program Comprehension and Change Propagation" (2001) [97]
Obstruction: Data Dealers
Expanse: Between Classes
Occurrence: Data
Hierarchies: Code Smell, Design Smell

Hidden Dependency is a situation where, inside a class, methods are silently resolving dependencies, hiding that behavior from the caller. *Hidden Dependencies* can cause a runtime exception when a caller has not set up the appropriate environment beforehand, which, by itself, is yet another code smell: *Required Teardown/Setup Code* [Section 4.1.9].

Causation:

Objects (which are not stateless) that require a constructor have an empty constructor - the essence of these objects should be passed on to creation, and even better if they are made immutable to avoid *Mutable Data* code smell.

Problems:

1. Coupling
   - Each Global Variable is basically a hidden dependency that a tester has to mock.
2. Hard to Test
   - Mocking is required to test methods that use data outside of its closest scope (class or parameters).
3. Error-Prone
   - Changing or removing data might unintentionally break code in unexpected places.
4. Decreased Comprehensibility
   - It is hard to understand the method's behavior without looking up the functions or variables outside its scope.

Refactor Methods:

1. Inject Dependencies.

```python
class Customer:
    pass

customer = Customer()

class Cart:
    def __init__(self):
        self.customer = customer  # gets customer from global scope

cart = Cart()
```

Listing 4.46. Hidden Dependencies Code Smell

```python
class Customer:
    pass

customer = Customer()

class Cart:
    def __init__(self, customer):
        self.customer = customer  # gets customer explicitly

cart = Cart(customer)
```

Listing 4.47. Hidden Dependencies Fix

4.5.3. Insider Trading

Known As: Inappropriate Intimacy
Origin: Fowler M., Kent B., "Refactoring: Improving the design of existing code" (1999) [56] as *"Inappropriate Intimacy"* renamed to *"Insider Trading"* in Fowler M., "Refactoring: Improving the Design of Existing Code" (2018) [26]
Obstruction: Data Dealers
Expanse: Between Classes
Occurrence: Responsibility
Hierarchies: Code Smell

The classes should know as little as possible about each other. As Fowler put it, "classes spend too much time delving in each other's private parts" [56]. This code smell was listed in 1999 as *Inappropriate Intimacy* and is no longer listed in the 2018 version of the book under the same name. It was updated by the term *Insider Trading* code smell, possibly to make the change - that now the `classes` were generalized to modules - more noticeable (similarly to the wording change in *Feature Envy* [Section 4.3.4]).

The concept stays the same - instead of reaching for each other's secrets, modules/classes interchange too much information and implementation details. In other words, this occurs whenever a module/class has too much knowledge about the inner workings or data of another module/class.

Causation:

At first, two classes could intertwine, but over time, they have coupled.

Problems:

1. Coupling
   - The modules between themselves should know as little as possible.
2. Reduced Reusability
   - Developers cannot reuse intertwined classes in isolation.
3. Hard to Test
   - Mocking is required.

Refactor Methods:

1. Move Method,
2. Move Field,
3. Encapsulate Field,
4. Replace Inheritance with Delegation,
5. Change Bidirectional Association to Unidirectional.

```python
@dataclass
class Commit:
    name: str

    def push(self, repo: Repo):
        repo.push(self.name)

    def commit(self, url: str):
        ...


@dataclass
class Repo:
    url: str

    def push(self, name: str):
        ...

    def commit(self, commit: Commit):
        commit.commit(self.url)
```

Listing 4.48. Insider Trading Code Smell

4.5.4. Message Chain

Origin: Fowler M., Kent B., "Refactoring: Improving the design of existing code" (1999) [56]
Obstruction: Data Dealers (Encapsulators)
Expanse: Between Classes
Occurrence: Message Calls
Hierarchies: Code Smell, Design Smell

Suppose that class `A` requires data from class `D`, but to retrieve those data, it has to make unnecessary calls to class `B` sequentially and then `C` to get it. This function sequencing is called Message Chain code smell. Long sequences of methods calls indicate hidden dependencies by being intermediaries. A sequence of temporary variables could have also hidden the sequence of methods. The problem with this smell is that any change in the intermediate relationship causes the client to have to change [26].

Causation:

Classes ask the objects to do the manipulation instead of telling the object with which manipulation should be done.

Problems:

1. Law of Demeter Principle Violation
   - Law of Demeter specifies that each class should have limited knowledge about other classes and only about these classes, which are "closely" related to the current class.
2. Tell, Don't Ask Principle Violation
   - The manipulation should be done by telling the object to manipulate, not by asking for permission to manipulate.

Refactor Methods:

1. Hide Delegate,
2. Extract Method,
3. Move Method.

Exceptions:

In the context of the system as a whole, some communication between modules must take place. All possibilities should be properly balanced so that none of the smells dominate *Global Data* [Section 4.5.1], *Tramp Data* [Section 4.5.6], Message Chain [Section 4.5.4], Middle Man [Section 4.5.5], to make the entire codebase as straightforward as possible.

```
class Minion:
    _location: Location

    def action(self):
        ...
        if self._location.field.is_frontline():
            ...

class Location:
    field: Field

class Field:
    def is_frontline(self)
        ...
```

Listing 4.49. Message Chain Code Smell

```
class Minion:
    _location: Location

    def action(self):
        ...
        if self.is_frontline():
            ...

    def is_frontline(self)
        return self._location.is_frontline()


class Location:
    _field: Field

    def is_frontline(self)
        return self._field.is_frontline()


class Field:
    def is_frontline(self)
        ...
```

Listing 4.50. Message Chain Fix

4.5.5. Middle Man

Origin: Fowler M., Kent B., "Refactoring: Improving the design of existing code" (1999) [56]
Obstruction: Data Dealers (Encapsulators)
Expanse: Between Classes
Occurrence: Message Calls
Hierarchies: Code Smell, Design Smell

The class that only performs delegation work to other classes is called a *Middle Man*. This is the opposite of the *Message Chains* [Section 4.5.4]. Encapsulation (hiding internal details) in the world of Object-Oriented Programming is a typical pattern [51]. However, the problem arises when it goes too far - Fowler specified that it could be said that it's terrible when half of the methods are delegators [26]. Mäntylä wrote that this is a problem when every time a new method has to be created, it requires the delegators to be modified with them [51].

Causation:

This can happen due to over-zealous refactorization of *Message Chains*.

Problems:

1. Readability
   - Without proper, meaningful, unambiguous naming for the delegation method, the developer might need to check what is being called to be sure.
2. Unnecessary Indirection
   - Holding references instead of actual values might slightly increase project complexity in volume.

Refactor Methods:

1. Remove Middle Man,
2. Inline Method,
3. Replace Delegation with Inheritance,
4. Replace Superclass with Delegate.

Exceptions:

In the context of the system as a whole, some communication between modules must take place. All possibilities should be appropriately balanced so that none of the smells dominate *Global Data* [Section 4.5.1], *Tramp Data* [Section 4.5.6], Message Chain [Section 4.5.4], Middle Man [Section 4.5.5], to make the entire codebase as straightforward as possible.

```python
class Minion:
    _location: Location

    def action(self):
        ...
        if self.is_frontline():
            ...

    def is_frontline(self)
        return self._location.is_frontline()


class Location:
    _field: Field

    def is_frontline(self)
        return self._field.is_frontline()


class Field:
    def is_frontline(self)
        ...
```

Listing 4.51. Middle Man Code Smell

```python
class Minion:
    _location: Location

    def action(self):
        ...
        if self._location.field.is_frontline():
            ...

class Location:
    field: Field

class Field:
    def is_frontline(self)
        ...
```

Listing 4.52. Middle Man Fix

4.5.6. Tramp Data

Origin: Steve Smith "Refactoring Fundamentals" (2013) [83], previously mentioned as Reason to Refactor by Steve McConnell in "Code Complete" (1993) [50]
Obstruction: Data Dealers
Expanse: Between Classes
Occurrence: Data
Hierarchies: Code Smell, Design Smell

This is very similar to the *Message Chains* code smell [Section 4.5.4], but with the difference that there, the pizza supplier was ordered to go through a long chain of method calls. Here, the pizzeria supplier additionally delivers *"pizza"* through that long chain of method calls, with "pizza" present in each of the method parameters. This is an indicator of Dubious Abstraction code smell [Section 4.2.3] - the data that pass through long chains of calls, most likely at least one of the levels, are not consistent with the abstraction presented by each of the routine interfaces [58].

Causation:

This can happen due to a cheap way of *Global Data* [Section 4.5.1] code smell refactorization.

Problems:

1. Law of Demeter Principle Violation
   - The functionality should be as close to the data it uses as possible.

Refactor Methods:

1. Extract Method,
2. Hide Delegate,
3. Move Method.

Exceptions:

In the context of the system as a whole, some communication between modules must take place. All possibilities should be appropriately balanced so that none of the smells dominate *Global Data* [Section 4.5.1], *Tramp Data* [Section 4.5.6], Message Chain [Section 4.5.4], Middle Man [Section 4.5.5], to make the entire codebase as straightforward as possible.

```
    timer: int
    match: Match

    def start_turn():
        match(timer).field(timer).players(timer).troops(timer)
```

Listing 4.53. Tramp Data Code Smell

4.6. OBJECTS ORIENTED ABUSERS

The common theme between the smells in the *Object-Oriented Abusers* category is that they usually violate one of the principles of Object-Oriented Programming. The way they behave hurts the possibilities provided by Object-Oriented Design [51]. These smells could have been rational solutions in other programming styles, such as *repeated switching* in procedural programming. Some of the smells are caused by the lack of proper subclasses or the lack of standard interfaces for closely related classes.

4.6.1. Alternative Classes with Different Interfaces

Known As: Complex Method, God Method, Brain Method
Origin: Fowler M., Kent B., "Refactoring: Improving the design of existing code" (1999) [56]
Obstruction: Object-Oriented Abusers
Expanse: Between Classes
Occurrence: Duplication
Hierarchies: Code Smell, Design Smell

If two classes have the same functionality but different implementations, developers should merge them, or developers should extract a superclass to limit code duplication [Section 4.4.2]. This smell occurs whenever a class can operate on two alternative classes (for example, take `Zombie` and `Snowman`). However, the interface to these alternative classes is different - when it operates with ‘Zombie‘, it calls `hug_zombie()`, and with ‘Snowman‘, it has to call `hug_snowman()`.

Causation:

This may happen due to the oversight that a functionally equivalent class already exists or when two or more developers are working on code to handle a similar situation. However, they did not know about the other’s work — lack of abstract methods that enforce the common implementation method names.

Problems:

1. Don’t Repeat Yourself Principle Violation

- "DRY Principle" - as the name suggests - is aimed to reduce repetition of the same code implementations.

Refactor Methods:

1. Change Function Declaration,
2. Move Function,
3. Extract Superclass.

```
class Snowman(Humanoid):
    def hug_snowman():
        ...

class Zombie(Humanoid):
    def hug_zombie():
        ...
```

Listing 4.54. AI with DI Code Smell

```
class Snowman(Humanoid):
    def hug():
        ...

class Zombie(Humanoid):
    def hug():
        ...
```

Listing 4.55. AI with DI fix

4.6.2. Base Class Depends on Subclass

Known As: Base Classes Depending on Their Derivatives
Origin: Robert C. Martin, "Clean Code: A Handbook of Agile Software Craftsmanship" (2008) [55] as "Base Classes Depending on Their Derivatives."
Obstruction: Object-Oriented Abusers
Expanse: Between Classes
Occurrence: Interfaces
Hierarchies: Antipattern, Code Smell, Design Smell

The rule is that the child classes should be deployable independently from the parent class. That allows us to deploy the system in discrete and independent components.

When one of these subclasses is modified, the base class does not need to be redeployed as well. In this way, the impact of change is much smaller, proportionally to the maintenance effort [55]. This smell is closely related to the *Shotgun Surgery* [Section 4.2.6] code smell.

Problems:

1. Liskov Substitution Principle Violation
   - Base class and subclass should be interchangeable without breaking the logic of the program.

4.6.3. Conditional Complexity

Known As: Repeated Switching, Switch Statement, Conditional Complexity, Prefer Polymorphism to if/else or switch/case
Origin: First appeared in Fowler M., Kent B., "Refactoring: Improving the design of existing code" (1999) as "Switch Statement" [56], updated to "Repeated Switching" in Fowler M., "Refactoring: Improving the design of existing code (3rd edition)" (2018), named as "Conditional Complexity" by Steve Smith in "Refactoring Fundamentals" [83] (2013)
Obstruction: Object-Oriented Abusers
Expanse: Within a Class
Occurrence: Conditional Logic
Hierarchies: Code Smell, Design Smell, Implementation Smell

In data-oriented programming, the usage of `switch`-like statements (lengthy, cascading `if` statements or `switch/case`) should be relatively rare. One such switch usually executes code scattered around the code base and should usually be replaced with a polymorphism solution.

This smell was phrased initially as *Switch Statement* by Fowler and Beck back in 1999 [56]. One year later, Mäntylä noted that such a name is misleading since switch statements do not necessarily imply a code smell but rather in the situation when they are used instead of a viable polymorphism solution [51]. Fowler, 14 years later, in his newest book in 2018, changed the name, suggesting *Repeated Switching*, agreeing that, fortuitously due to the way he initially phrased it, conditional statements got a bad reputation, while he never unconditionally opposed the existence of all `if`-s and `switch`-es [26]. In the "Clean Code" by Robert Martin, the smell was called "Prefer Polymorphism to if/else or switch/case", which is lengthy, but it hits the nail on the head [55].

I provide a typical example of this issue on an `Exporter` class with different file formats. A new `elif` has been added for each new feature instead of using an Object-Oriented solution like the Factory Method.

Keeping a class focused on a single concern is vital to make it more robust. Martin Fowler defines responsibility as a reason to change, concluding that "A class should have only one reason to change." [26]. An example that violates this would be a class that prints a table that handles both the contents of the cells and the styling of the table.

Another situation, which is not explicitly mentioned in the sources, would be a nested `Try` and `Except/Catch` "checklist", where numerous error-catching blocks are used instead of one generalized for the situation at hand.

Causation:

The most common way developers can create such a smell is when a conditional switch behavior is used instead of creating a new class. The first time it happens is not yet a "scent-ish" smell, but this immediately becomes a saturated red flag as soon as it is done the second time [92]. Logic blocks grew over time more extensive, and no one bothered to implement an Object-Oriented Programming style alternative like decorator, strategy, or state. It was easier to add another `else if`.

Problems:

1. Open-Closed Principle Violation
   - The class should be open for extension but closed for modification
2. Comprehensibility
   - If statement blocks make things harder to understand as they require to think about all the possible paths the logic can go.
3. Unnecessary Indirection
   - Most likely, there is a way to execute one method on a polymorphic class instead of switching cases.
4. Increased Test Complexity
   - Each subsequent if branch needs each subsequent test.
5. More Complex APIs

Refactor Methods:

1. Use a Guard Clause,
2. Extract Conditional,
3. Replace with Polymorphism,
4. Use Strategy Pattern,

5. Use Null Object,
6. Use Functional Programming Based Solution.

```
class Exporter:
    def export(self, export_format: str):
        if export_format == 'wav':
            self.exportInWav()
        elif export_format == 'flac':
            self.exportInFlac()
        elif export_format == 'mp3':
            self.exportInMp3()
        elif export_format == 'ogg':
            self.exportInOgg()
```

Listing 4.56. Conditional Complexity Code Smell

```
class Exporter:
    def export(self, export_format: str):
        exporter = self.get_format_factory(export_format)
        exporter.export()
    def get_format_factory(self, export_format: str):
        if export_format in self.export_format_factories:
            return render_factory[export_format]
        raise MissingFormatException
            ...
```

Listing 4.57. Conditional Complexity Fix: Factory

4.6.4. Inappropriate Static

Known As: Static Cling
Origin: Robert Cecil Martin, "Clean Code: A Handbook of Agile Software Craftsmanship" (2008) [55] for "Inappropriate Static"
Obstruction: Object-Oriented Abusers
Expanse: Between Classes
Occurrence: Interfaces
Hierarchies: Code Smell, Design Smell

The rule of thumb given by uncle Robert is that when in doubt, one should prefer non-static methods to static methods. The best way to check whether a method should be static would be to think if the method should behave polymorphically. An excellent example of a static method given by Martin is `math.max(double a, double b)` - it is hard to think of polymorphic behavior for a `max` function. On the other hand, `HourlyPayCalculator.calculate_pay(employee, overtime_rate)` is dubious, as there could be different algorithms to calculate the payment, and thus it should be a nonstatic method of class `Employee`. However, one should be aware and take caution of the *Speculative Generality* Code Smell [Section 4.4.4]. At the very present moment, when there are no different algorithms yet requested or planned, this is a stateless operation, which is acceptable for static methods. Steve Smiths addresses that statics should be reserved for behavior that will never change, besides the previously mentioned stateless operations, giving global constants as examples [83].

Static Cling is a code smell based on the border of the test code and the source code, although, following the Fail-Fast principle, the issue starts in the developing parts of the codebase.

Whenever a static function is called, in most languages, it is, to say the least, not trivial to test or mock the method in which it occurs [Listing 4.58]. There are 3rd party mocking frameworks, but that is more of a workaround for bad design. Developers should look out for these dependencies because they are effortlessly introduced into the code in styles other than Test-Driven Development.

Problems:

1. Hard to Test
   - Using static functions and variables makes the code harder to test; requires mocking.
2. Coupling
3. Single Responsibility Principle Violation

Refactor Methods:

1. Encapsulate Method.

```
class FooUtils:
    @staticmethod
    def wrap_tag_premium(foo: Foo):
        return f"[TAG]: {foo.action()}"

    @staticmethod
    def wrap_tag_special(foo: Foo):
        return f"[TAG]: {foo.action()}"
```

Listing 4.58. Inappropriate Static Code Smell

```
@dataclass
class FooTagWrapper:
    foo: Foo

    def wrap_tag_premium(self):
        return f"[PREMIUM]: {self.foo)}"

    def wrap_tag_special(self):
        return f"[SPECIAL]: {self.foo)}"
```

Listing 4.59. Inappropriate Static Fix

4.6.5. Refused Bequest

Known As: Refused Parent Bequest
Origin: Fowler M., Kent B., "Refactoring: Improving the design of existing code" (1999) [56]
Obstruction: Object-Oriented Abusers
Expanse: Between Classes
Occurrence: Interfaces
Hierarchies: Code Smell, Design Smell

Whenever a subclass inherits from a parent but only uses a subset of the implemented parent methods, that is called *Refused Bequest*. This behavior can happen both implicitly and explicitly:

- Implicitly, when the inherited routine does not work
- Explicitly, if an error is thrown instead of supporting the method

Whenever a child class is created, it should fully support all the data and methods it inherits [51]. Fowler says that this smell is not that strong, though, and admits that he sometimes reuses only a bit of behavior, but it can cause confusion and problems [56].

However, there is one crucial thing. Both Fowler and Mäntylä are pointing out a strong case of this Code Smell whenever a subclass is reusing behavior but does not want to support the superclass interface [26, 35, 51].

Causation:

This could happen due to bad development decisions when part of the needed functionality is already implemented in another class, but in general, the parent class is about something different from whatever the developer would like to implement in the new class. This inconsistency most likely indicates that the inheritance does not make sense, and the subclass is not an example of its parent [92].

Problems:

1. Liskov Substitution Principle Violation
   - The *Liskov Substitution Principle* describes that the relationship of subtypes and supertypes should ensure that any property proved about supertype object also holds for its subtype object.
2. Hard to Test
   - Each subclass might have different capabilities up and down the class hierarchies.

Refactor Methods:

1. Extract Subclass,
2. Push Down Field,

```
class Minion(ABC):
    @abstractmethod
    def attack(self):
        """ """

    @abstractmethod
    def move(self):
        """ """

class Tower(Minion):
    def attack(self):
        ...

    def move(self):
        raise NotImplementedException
```

Listing 4.60. Refused Bequest Code Smell

3. Push Down Method,
4. Replace Inheritance with Delegation,
5. Replace Superclass/Subclass with Delegate.

4.6.6. Temporary Field

Origin: Fowler M., Kent B., "Refactoring: Improving the design of existing code" (1999) [56]
Obstruction: Object-Oriented Abusers
Expanse: Within a Class
Occurrence: Data
Hierarchies: Code Smell

Temporary Field is a variable created where it is not needed. It refers to variables only used in some situations [51] or specific areas of a program. This uniqueness can be confusing when the purpose of using the variable cannot be explained or cannot be found outside of its scope [35]. It might be misplaced at the class level when the functionality it provides is specific only to a particular method [35]. One should expect an object to need all of its fields.

Problems:

1. Hard to Understand
   - Additional fields clutter the code and strain the cognitive load by keeping in mind useless attributes.

Refactor Methods:

1. Introduce Null Object,
2. Extract Class,
3. Move Function.

```python
@dataclass
class MyDateTime:
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day
        self.full_date = f"{year}, {month}, {day}"

    def foo(self):
        ...

    def goo(self):
        ...

    def hoo(self):
        ...

    def __str__(self):
        return self.full_date
```

Listing 4.61. Temporary Field Code Smell

```python
@dataclass
class MyDateTime:
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

    def foo(self):
        ...

    def goo(self):
        ...

    def hoo(self):
        ...

    def __str__(self):
        return f"{self.year}, {self.month}, {self.day}"
```

Listing 4.62. Temporary Field Fix

4.7. FUNCTIONAL ABUSERS

Functional programming brought many potent and valuable tools and techniques that developers can use in object-oriented programming. Functional programming significantly increases the reliability of the software. One of the key concepts is the lack of side effects, the writing of the so-called "pure functions". The immutability of data is an equally important concept. Apart from the reverse of those two concepts of code smells (so: *Mutable Data* and *Impure Functions* or *Side Effects*), there is also the *Boolean Blindness* [Section 4.8.1] Code Smell primarily prevalent among the Haskell community, which is listed in the *Obfuscators* category and *Imperative Loops* [Section 4.7.1] list under the *Bloaters* category. Functional programmers are iterating with `forEach`, `pipes`, or `streams`, which are much better tools than the error-prone imperative loop.

4.7.1. Imperative Loops

Known As: Explicitly Indexed Loops, Indexed Loops, Loops
Origin: Fowler M., Kent B., "Refactoring: Improving the design of existing code" (2018) as "Loops" [56], renamed by Jerzyk M. to "Imperative Loops" in "Code Smells: A Comprehensive Online Catalog and Taxonomy" (2022)
Obstruction: Functional Abusers
Expanse: Within a Class
Occurrence: Unnecessary Complexity
Hierarchies: Code Smell

Martin 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 [56]. 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` [26].

Indeed, loops can sometimes be hard to read and error-prone. This might be unconfirmed, but I 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* [Section 4.9.1] and check if there is already a built-in function that will take care of the desired operation.

I 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 [1] or Python [2]. 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 [Section 4.1.4] or Side Effects [Section 4.7.3] code smells.

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

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.

Causation:

It is impossible to easily overwrite the information given in the old books or video tutorials, which was pretty standard due to the lack of any other alternatives. People have learned that type of looping and may not even suspect that there are alternatives until they find them. Similarly, developers who come from older languages, which do not yet offer such facilities, can use explicit iteration habitually.

Problems:

1. Readability
   - In contrast to pipelines, loops don't provide declarative readability of what is precisely being processed.

Refactor Methods:

1. Replace Loop with Pipeline,
2. Replace Loop with built-in.

```
const examples = ["foo", "bar", "baz"];
for (let idx = 0; idx < examples.length; idx++) {
  console.log(examples[idx]);
}
```

Listing 4.63. Explicitly Indexed Loop Code Smell

```
const examples = ["foo", "bar", "baz"];
examples.forEach((example) => console.log(example));
```

Listing 4.64. Explicitly Indexed Loop Fix

```
const examples = ["foo", "bar", "baz"];
let bar_in_examples = false;
for (let idx = 0; idx < examples.length; idx++) {
  if (examples[idx] == "bar") {
    bar_in_examples = true;
  }
}
console.log(bar_in_examples); // true
```

Listing 4.65. Loop and Clever Code Code Smell

```
const examples = ["foo", "bar", "baz"];
console.log(examples.includes("foo")); // true
```

Listing 4.66. Loop and Clever Code Fix

4.7.2. Mutable Data

Origin: Fowler M., Kent B., "Refactoring: Improving the Design of Existing Code" (2018) [56]
Obstruction: Functional Abusers
Expanse: Between Classes
Occurrence: Data
Hierarchies: Code Smell

Mutable data are harmful because they can unexpectedly fail other parts of the code. It is a rich source of bugs that are hard to spot because they can occur under rare conditions. Fowler says that this is a significant factor that contributed to the rise of the new programming school, functional programming, in which one of the principles is that the data should never change. While these languages are still relatively small, he states that developers should not ignore the advantages of immutable data [26]. It is hard to reason about these variables, especially when they have several reassignments.

Causation:

In Object-Oriented Programming, the immutability of objects was not addressed audibly, if at all, in the context of something desirable. This is a relatively new concept.

Problems:

1. Error-Prone
   - Mutable Data is a rich source of hidden bugs that might be spotted only when specific rare conditions are met.
2. Hard to Understand

- Data that might change at any moment is hard to reason about without excluding every corner case.

```python
@dataclass
class Foo:
    name: str
    value: float
    premium: bool

# foo object instance will be passed around and modified
```

Listing 4.67. Mutable Data Code Smell

```python
@dataclass(frozen=True)
class Foo:
    name: str
    value: float
    premium: bool

# foo object instance will be passed around and, if that is necessary,
# recreated, thus making the state change explicit for everyone and
↪   handled
```

Listing 4.68. Mutable Data Fix: Making the attributes immutable

Refactor Methods:

1. Remove Setting Method,
2. Choose Proper Access Control,
3. Separate Query from Modifier,
4. Change Reference to Value,
5. Replace Derived Variable with Query,
6. Extract Method,
7. Encapsulate Variable,
8. Combine Methods into Class.

4.7.3. Side Effects

Known As: Impure Functions
Origin: Marcel Jerzyk, "Code Smells: A Comprehensive Online Catalog and Taxonomy" (2022)
Obstruction: Functional Abusers
Expanse: Within a Class
Occurrence: Responsibility
Hierarchies: Code Smell

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. 4.69] 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 triggers 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. I 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 [cs. 4.3.2] and another possible bad solution, `set_gold(amount: int, is_payday: bool)`, would cause *Flag Arguments* Code Smell [Section 4.2.4].

Causation:

Developers added additional actions to existing functionalities, which usually took place in the context during typical use and are not explicitly related to the function itself.

Problems:

1. Single Responsibility Principle Violation
   - Method is literally doing more than one thing.

Refactor Methods:

1. Extract Method,
2. Extract Field.

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

Listing 4.69. Seemingly Readable Code Snippet

```
@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 = find_player_by_name(Marcel, Jerzyk)
marcel.set_gold(0)
marcel.set_health(Health.Decent)
```

Listing 4.70. Side Effects Code Smell

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

    def payout_routine(self):
        self.trigger_animation(Animation.Dancing)
        self.job.reset_payday_timer()

    def set_gold(self, amount: int):
        self.gold = amount

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

Listing 4.71. Side Effects Fix

4.8. LEXICAL ABUSERS

This category highlights something that I would like to pay special attention to when researching code smells - comprehensibility. Anything that overuses the human ability to understand the reading text - or the other way around - anything that does not provide any additional verbal value that a human could use to learn about a program's operation is placed within the *Lexical Abusers* subcategory. Properly selected names in the code should accelerate the speed of understanding or even make it possible to skip reading blocks of code entirely, solely based on properly formulated signatures.

4.8.1. Boolean Blindness

Origin: Marcel Jerzyk, "Code Smells: A Comprehensive Online Catalog and Taxonomy" (2022), noted in [16], [95], [90]. Robert Harper attributes this term to Dan Licata [36]
Obstruction: Lexical Abusers
Expanse: Within a Class
Occurrence: Names
Hierarchies: Code Smell

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` [Listing 4.72]? *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 [Section 4.8.6] and Magic Numbers [Section 4.8.5].

Problems:

1. Comprehensibility
   - Neither in real life one can answer just *yes/no* without ever confusing interlocutor to every single closed question that may possible.

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

Listing 4.72. Ambiguity of Boolean

Refactor Methods:

1. Introduce New Type.

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

Listing 4.73. Boolean Blindness Fix: Meaningful Boolean Type

4.8.2. Fallacious Comment

Known As: Comment, Attribute Signature and Comment are Opposite; Method Signature and Comment are Opposite
Origin: Marcel Jerzyk, "Code Smells: A Comprehensive Online Catalog and Taxonomy" (2022), noted as two different Lexical Antipatterns by Arnaoudova in "Linguistic Antipatterns: What they are and how developers perceive them" (2015) [5]
Obstruction: Lexical Abusers
Expanse: Within a Class
Occurrence: Names
Hierarchies: Code Smell, Linguistic Antipattern

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* [Section 4.4.5] are a Code Smell themselves, and so is Duplicated Code [Section 4.4.2].

This might generally happen within docstrings in real-life scenarios, which developers usually find in methods exposed to other users.

Causation:

The developer was in a hurry and did not double-check that everything was up-to-date after the changes. A passing unit test could also reaffirm him - there is no practical automated way to check for the correctness of comments/docstrings.

Problems:

1. Decreased Readability
   - The developer does not know whether he should trust the method's signature or comment.

Refactor Methods:

1. Remove Inconsistency.

```python
def rename_description(product, manufacturer):
    """
    Adds the manufacturer footer to the
    products description.
    """
    ...
```

Listing 4.74. Fallacious Comment Code Smell

```python
def add_manufacturer_footer_to_product_description(product, manufacturer):
    ...
```

Listing 4.75. Fallacious Comment Fix: Method Rename

4.8.3. Fallacious Method Name

Known As: "Set" Method returns, "Get" Method does not Return, "Get" Method - More than an Accessor, "Is" Method - More than a Boolean, Expecting but not getting a collection, Expecting but not getting a single instance, Not answered question, Validation method does not confirm, Use Standard Nomenclature Where Possible
Origin: Marcel Jerzyk, "Code Smells: A Comprehensive Online Catalog and Taxonomy" (2022), noted nine different Lexical Antipatterns by Arnaoudova in "Linguistic Antipatterns: What they are and how developers perceive them" (2015) [5]
Obstruction: Lexical Abusers
Expanse: Within a Class
Occurrence: Names
Hierarchies: Code Smell, Implementation Smell, Linguistic Antipattern

When I started to think of Code Smells from the comprehensibility perspective (of its lack of) as one of the critical factors, I was 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* [Section 4.9.5] or code without any explanation (*Magic Number* [Section 4.8.5], *Uncommunicative Name* [Section 4.8.6]). I felt like there was a missing hole in the code that was intentionally too clever (*Clever Code* [Section 4.9.1]) or the code that contradicts itself. Fortunately, I have found a fantastic article supporting my thoughts and addressing some of what I had in mind under the name *Linguistic Antipatterns* [5]. I have included their subset of antipatterns under one code smell because listing them one by one would be too granular from a code perspective. Their idea 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.

Causation:

When we create methods, their name may not entirely represent what they will be doing after we finish working on them. Except that one should then go back and correct its name accordingly to what has been done.

Problems:

1. Decreased Readability
   - Developers need to inspect the function or methods in order to find out they do.
2. Reduced Reliability
   - If someone would like to use a method with a given name, he should also expect the name's effect.

Refactor Methods:

1. Remove Contradictions,
2. Rename Method.

```
def getFoos() -> Foo:
    ...
    return Foo

def isGoo() -> str:
    ...
    return 'yes'

def setValue(self, value) -> Any:
    ...
    return new_value
```

Listing 4.76. Fallacious Method Name Code Smells

```
def getFoos() -> list[Foo]:
    ...
    foos: list[Foo] = ...
    return list[Foo]

def isGoo() -> bool:
    ...
    return True

def setValue(self, value) -> None:
    ...
```

Listing 4.77. Fallacious Method Name Fix

4.8.4. Inconsistent Names

Known As: Use Standard Nomenclature Where Possible
Origin: William C. Wake, "Refactoring Workbook" (2004) [92]
Obstruction: Lexical Abusers
Expanse: Within a Class
Occurrence: Names
Hierarchies: Code Smell, Implementation Smell, Linguistic Antipattern

Human brains work in a pattern-like fashion. Starting from the first class, the concept of operation and use of each subsequent class should be generalized throughout the project, facilitating and iteratively accelerating the speed of understanding how the code works.

For this reason, once we know that one class uses the method `store()`, we should expect that another class for the very same mechanic also uses the `store()` name for that, instead of `add()`, `put()` or `place()`.

Causation:

In a team project, members could have omitted checking the existing naming in other classes [92]. They could also intentionally choose different naming methods to distinguish classes according to the naming convention of functions.

Problems:

1. Comprehensibility
   - Standardized communication through names is vital for mental shortcuts.
2. Flow State Disruption
   - The developer expects a method inside a sibling class but can't find it and has to look up synonymous variations of the method he wants.

Refactor Methods:

1. Rename Method.

```
class Human:
    def talk():
        ...

class Elf:
    def chat():
        ...
```

Listing 4.78. Inconsistent Names Code Smell

```
class Character(ABC):
    @abstractmethod
    def talk():
        """ Converse """

class Human(Character):
    def talk():
        ...

class Elf(Character):
    def talk():
        ...
```

Listing 4.79. Inconsistent Names Fix

4.8.5. Magic Number

Known As: Uncommunicative Number
Origin: William C. Wake, "Refactoring Workbook" (2004) [92]
Obstruction: Lexical Abusers
Expanse: Within a Class
Occurrence: Names
Hierarchies: Antipattern, Code Smell, Implementation Smell

The problem with floats and integers is that they convey no meaning - there is only context, which is often not enough, and even if one might think it is, leaving the unexplained number makes the code just less readable. Magic numbers also usually go in small groups, which should encourage one to collect them under one appropriately named constant.

Causation:

It may be convenient to write the implementation idea first with numbers, but then, after a task is finished, the developer did not get back to add meaning to the used abstraction.

Problems:

1. Low Readability
   - Only funny numbers convey universally understood meaning; in all other cases - they are just digits without explanation.
2. Duplication
   - If the place where the magic number is placed is not a Query to some database, then more likely than not, programmers will repeat this number somewhere later in the logic.
3. Bijection Violation
   - A number as a model is not in one to one relationship with the domain (5 can mean *5 years*, *5 apples*, *grade type*, *5 Bitcoins*, ...).

```
def calculateDamage(...) -> int:
    total_damage = ...
    return math.min(100, damage)
```

Listing 4.80. Magic Number Code Smell

```
def calculateDamage(...) -> int:
    total_damage = ...

    MAX_DAMAGE_CAP: int = 100
    return math.min(MAX_DAMAGE_CAP, total_damage)
```

Listing 4.81. Magic Number Fix: Symbolic Constant

Refactor Methods:

1. Replace with Symbolic Constant,
2. Replace with Parameter.

Exceptions:

Developers should not always replace numbers with intentional names. The best example that I can give is math & physics formulas. The formula for the kinematic energy would be written as `kinematic_energy = (mass * (velocity**2))/2`, leaving *two's* as is. On the other hand, if a formula had a known constant like `PI`, then I would create a constant that expresses that, in fact, `3.14159` is `PI` (it also could be considered as a *Clever Code* [Section 4.9.1], it would be best to use `math.pi` built-in instead).

4.8.6. Uncommunicative Names

Origin: William C. Wake, "Refactoring Workbook" (2004) [92]
Obstruction: Lexical Abusers
Expanse: Within a Class
Occurrence: Names
Hierarchies: Code Smell, Implementation Smell

The name should convey meaning and meaning that is preferably not misleading (Fallacious Method Name [Section 4.8.3]. Descriptive names can save countless hours if they are good enough, just like a good abstract in a scientific article. The code should be as expressive as possible [19]. In the "Clean Code" by Robert Martin, this smell is "shredded" into five very descriptive smells and recommendations that underline the importance of having "good labels" [55]:

1. *Obscured Intent* [Section 4.9.5],
2. Function Names Should Say What They Do,
3. Choose Descriptive Names,
4. Unambiguous Names,
5. Names Should Describe Side-Effects

Martin Fowler added this smell under the name *"Mysterious Name"* in his third edition of the Refactoring book, saying that a good name, with much thought put into its definition, can save hours of incomprehensibility problems later. He says that titles should communicate what they do and how to use them.

Causation:

People tend not to return to the names of the variables or methods that they have already declared. Usually, it is the best they can come up with at the declaration moment, but maybe later on, there could have been a much better name for the thing thought of. The names could also be too short, and the developer could be afraid of making them longer, thus cutting off the meaning potential.

Problems:

1. Comprehensibility
   - Good names are one of the most critical factors contributing to the Clean Code feeling. If the developer can not rely on the naming of variables, methods, and classes, it drastically reduces his ability to understand everything.

Refactor Methods:

1. Change Method Declaration,

2. Rename Variable,
3. Rename Field.

```
data = m1.get_f()
data_2 = m2.get_f()

value = data_2['dmg'] * data['def']
val = math.rand(value-3, value+3)
```

Listing 4.82. Uncommunicative Name Code Smell

```
def wobble_the_value(value: int, wobble_by: int):
    """ Adds tiny bit of randomness to the output """
    return math.rand(value-wobble_by, value+wobble_by)

attack_information: FightingInformation =
↪   attacking_minion.get_fighting_information()
defense_information: FightingInformation =
↪   defending_minion.get_fighting_information()

calculated_damage: int = attack_information.damage *
↪   defense_information.defense
final_damage_dealt: int = wobble_the_value(calculated_damage, wobble_by=3)
```

Listing 4.83. Uncommunicative Name Fix

4.9. OBFUSCATORS

Picking good words is hard. Humans often find themselves angry for choosing the wrong words or sentences in recent past situations, often finding the right ones after the event of their worries. The trouble with programming is not different. Sometimes, names are obscure and hinder the meaning, reducing the comprehensibility of the code. However, it also works the other way round - a good name for a method can be such a good metaphor or explanation that the code it contains is no longer required to be read to understand what the method does. Wake wrote that picking a good name is worth the effort [92]. The *Obfuscators* category contains smells that directly reduce the understandability of the code.

4.9.1. Clever Code

Origin: Marcel Jerzyk, "Code Smells: A Comprehensive Online Catalog and Taxonomy" (2022)
Obstruction: Obfuscator
Expanse: Within a Class
Occurrence: Unnecessary Complexity
Hierarchies: Code Smell

I am creating a conscious distinction between the *Obscured Intent* [Section 4.9.5] and *Clever Code*. While Obscured Intent addresses the ambiguity of implementation, emphasizing the incomprehensibility of a fragment of code, the Clever Code, on the other hand, can be confusing even though it is understandable.

Things that fall into this smell are codes that do something strange. This smell 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. Developers can find examples of both in the first example provided [Listing 4.84] - the code is reinventing the wheel of calculating the length of a string, which is one case of *"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. At the same time, on a day-to-day basis, he is a C language developer and did not know about `len()`. The other case might 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* [Section 4.10.1]). I 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. This re-implantation might cause problems if the implementation is not correct, or even if it is, perhaps there can be a performance hit compared to using the standard built-in option. This 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 strain the cognitive load required to process it. It could be classified as *Clever Code* (also emphasizing the ironic side of this saying). However, it fits more closely with the definition of the Complicated Boolean Expression [Section 4.9.2], and Binary Operator In Name [Section 4.3.2].

Causation:

There is a natural tendency to show off the new skills or techniques that one has learned, which are unnecessary in the current part of code that one is currently working in. In another case, maybe someone implemented their variation of a built-in or popular library instead of just using the tools given as built-ins or are a popular solution. There is also the same affinity as with *Obscured Intent* [Section 4.9.5] - sometimes, a developer can forget that something that seems obvious to him is not as clear to other developers. There may also be cases where the developer intentionally compacts the code to appear brighter by making it more mysterious.

Problems:

1. Comprehensibility
   - There is no benefit to making other developers required to rediscover/relearn things already built-in and well recognized.
2. Keep It Simple Stupid Principle Violation
   - There's no need to obfuscate the code with new personal creations when ready-made solutions are available.
3. Don't Repeat Yourself Principle Violation
   - Reimplemented "wheel" requires people delegated to maintain it or extend it instead, which is worse than just using whatever "the factory" has already given.

Refactor Methods:

1. Replace with Built-In,
2. Replace with Library.

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

message = 'Hello World!'
message_length = get_length_of_string(message)
print(message_length) # 12
```

Listing 4.84. Clever Code Code Smell: Reimplementation of Built-In

```python
message = 'Hello World!'
message_length = len(message)
print(message_length) # 12
```

Listing 4.85. Clever Code Fix: Use the Built-In

```python
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)
```

Listing 4.86. Clever Code Code Smell: Reimplementation of the Standard Library

```python
from collections import defaultdict
```

Listing 4.87. Clever Code Fix: Use the Standard Library

4.9.2. Complicated Boolean Expression

Origin: William C. Wake, "Refactoring Workbook" (2004) [92]
Obstruction: Obfuscator
Expanse: Within a Class
Occurrence: Unnecessary Complexity
Hierarchies: Code Smell

One should keep in mind that *Boolean Expressions* can be troublesome for some people (in other words, not as quick to comprehend as if it were an adequately named method instead). Wake reminds the reader of *De Morgan Laws* [3] that could be applied to simplify the expression.

Wake also reminds us about guard checks, which could "peel off" complexity layers from the expression [92]. Interestingly, Martin gives a minimum example [Listing 4.88] in his *Encapsulate Conditionals* refactoring explanation [55], which could be a bit controversial but is yet another good perspective to look at the problem. Instead of simplifying the expression, he paid attention to the readable intention. As I have just mentioned, the example provided might be a bit controversial because it contains just one `AND` operation, which is not a *Discrete Mathematics 2* type problem. Nevertheless, the point is valid: An appropriately named function or method is more comprehensible at a glance than whatever the `boolean` expression it is.

I have linked the smell to *Obscured Intent* [Section 4.9.5] as *causes* just because of all the cases of any `if not thing.notDone()` double negated by token and name expressions, wherever they are. An unnecessary *Flag Argument* [Section 4.2.4] Code Smell might imperceptibly impact the complexity of an expression.

Causation:

Similar to *Conditional Complexity* [Section 4.6.3], - developers introduced new features, and instead of doing it cleanly, developers used a dirty method conditional method instead.

Problems:

1. Comprehensibility
   - It's easier to read declarative words than to computate logic statements.

Refactor Methods:

1. Introduce Explaining Method or Variable,
2. Use Guard Clauses,
3. Simplify Conditional.

[3] De Morgan's Laws - Pair of transformation rules in boolean algebra. $\neg(A \vee B) \iff (\neg A) \wedge (\neg B)$ and $\neg(A \wedge B) \iff (\neg A) \vee (\neg B)$

```python
if (timer.has_expired() and not timer.is_recurrent()):
    ...
```

Listing 4.88. Complicated Boolean Expression Code Smell (Fowler example)

```python
if (should_be_deleted(timer)):
    ...
```

Listing 4.89. Complicated Boolean Expression Fix (Fowler example)

```python
def cook(ready: bool, bag: list):
    if (ready):
        if (['raspberry', 'apple', 'tomato'] in bag and ['carrot',
        ↪  'spinach', 'garlic'] not in bag):
            ...
```

Listing 4.90. Complicated Boolean Expression Code Smell

```python
# "ready" extracted out of the function scope
def cook(bag: list):
    def hasFruit(container: list) -> bool:
        return ['raspberry', 'apple', 'tomato'] in container
    def hasVeggie(container: list) -> bool:
        return ['carrot', 'spinach', 'garlic'] in container

    if not hasFruit(bag):
        return
    if hasVeggie(bag):
        return
    ...
```

Listing 4.91. Complicated Boolean Expression Fix

4.9.3. Complicated Regex Expression

Origin: Marcel Jerzyk, "Code Smells: A Comprehensive Online Catalog and Taxonomy" (2022)
Obstruction: Obfuscator
Expanse: Within a Class
Occurrence: Names
Hierarchies: Code Smell

Two bad things can be done that I 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* [Section 4.9.2], only that the human population it affects is much larger - there are more people who will quickly catch the meaning behind Boolean logic, but far 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, 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 [12].

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 [Section 4.9.5], should work just fine as they are. Getting a standardized and verified regular expression is okay. However, if one has the urge to create his own for his particular needs, then the developer should take care to break it down nicely, so any other developer does not need to debug a collection of squeezed characters.

Problems:

1. Comprehensibility
   - It's easier to read declarative words than to compute logic statements

Refactor Methods:

1. Extract Method,
2. Extract Variable.

```
regex_pattern = '(\W|^)(\w*)\s-\s[0-9]?[0-9]:[0-9][0-9]'
```

Listing 4.92. Complicated Regex Expression Code Smell

```
def get_regex_pattern_current_city_time() -> str:
    """
    Example Match:
      - `Wroclaw - 17:42`
      - `Berlin - 17:42`
      - `San Jose - 10:42`

    Compiled: (\W|^)(\w*)\s-\s[0-9]?[0-9]:[0-9][0-9]
    """
    prevent_excessive_match = '(\W|^)'
    city = '(\w*)'
    indication = '\s-\s'
    hour = '[0-9]?[0-9]'
    minute = '[0-9][0-9]'
    time = f'{hour}:{minute}'
    return f"{prevent_excessive_match}{city}{indication}{time}"
```

Listing 4.93. Complicated Regex Expression Fix: Variable Explanations

4.9.4. Inconsistent Style

Origin: Marcel Jerzyk, "Code Smells: A Comprehensive Online Catalog and Taxonomy" (2022)
Obstruction: Obfuscator
Expanse: Between Classes
Occurrence: Names
Hierarchies: Code Smell

The same thing as in *Inconsistent Names* [Section 4.8.4] applies to the general formatting and code style used in the project. Browsing through the code should have a similar feeling 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 so as not to disturb the expected form of code in the following lines as shown in Listing 4.94.

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*, for example, in the order of parameters within classes or methods. Once defined, the order should be kept in the group of all abstractions on that particular subject. If the order is not preserved, it leads, once again, to the unpleasant feeling of dissatisfaction after (if ever!) the mind realizes that it was again surprised wrong. [Listing 4.95] Depending on the specific case, it would still be only half the trouble if the flipped parameters were of different types (such as `string` and `int`). If the type were the same (e.g., `int`), this could lead unnoticeably to a significant hidden bug.

Causation:

Team members working on the same project disagreed on one particular coding style, linter, or code formatter. In the worst case, different people could use different formatters or different formatting rules and overwrite the whole files with their style of choice over each other, littering the git history with new commits.

Problems:

1. Error-Prone
   - With advanced IDE type-hinting nowadays, change is smaller, but when a bug gets introduced by a sequence inconsistency, it might not be enjoyable to find out its root cause.
2. Comprehensibility
   - Depending on the state of code, the comprehensibility issues that the inconsistent style can cause range from irrelevant up to illegible.
3. Flow State Disruption
   - Familiarity is an essential factor in code orientation and navigation.

Refactor Methods:

1. Introduce Linter Rules,
2. Reorder Parameters.

```
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 4.94. Inconsistent Style Code Smell:

```
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)  # potentially overlooked error
```

Listing 4.95. Inconsistent Style Code Smell: Sequence Inconsistency

4.9.5. Obscured Intent

Origin: Robert Cecil Martin, "Clean Code: A Handbook of Agile Software Craftsmanship" (2008) [55]
Obstruction: Obfuscators
Expanse: Between Classes
Occurrence: Unnecessary Complexity
Hierarchies: Code Smell

The Obscured Intent code smell, initially proposed by Martin in *"Clean Code"* [55], is cross-smelly with other smells in the *Obfuscators* category. When *Uncommunicative Names/Numbers* [Section 4.8.6]/[Section 4.8.5] are placed within a Vertically Separated Space [Section 4.9.7] containing an Imperative Loop [Section 4.7.1] with a *"What" Comment* [Section 4.4.5] explanation on top of that, then the *Obscured Intent* is going to be caught quite easily by intuition or by metrics of other smells.

The code should be as expressive as possible [55]. Martin gives an example Listing 4.96 of an utterly unreadable overtime function noting that even if the code is small and compact, it may still be tragic. In this case, correcting the *Uncommunicative Names/Numbers* would probably do the trick to make that snippet of code much more fragrant.

There is a famous real-world example of an Obscured Intent. In the *Quake 3: Arena fast inverse square root* Listing 4.97, the problem is with the *Uncommunicative Naming*, *"What" Comments*, *Dead Code*, and *Magic Numbers*. I will also point out that games have a slightly different specificity for their industry - usually, the code out there is very confusing, and priorities are not put on things such as reusability, so there is a lot to explore.

Causation:

Sometimes, a developer can forget that something that seems evident to him is not as clear to other developers. There may also be cases where the developer intentionally compacts the code to appear brighter by making it more mysterious.

Problems:

1. Comprehensibility Issues
   - Code does not convey meaning and thus is not understandable. It may be a considerable time waste if someone ever has to touch parts of the code that interact with an *obscure intent* piece of code.

Refactor Methods:

1. Remove the Code Smells that cause Obscure Intent.

```python
def m_ot_calc() -> int:
    return i_ths_wkd * i_ths_rte \
        + round(0.5 * i_ths_rte *
        max(0, i_ths_wkd - 400))
```

Listing 4.96. Obscured Intent Code Smell (Martin Example)

```c
float Q_rsqrt( float number )
{
        long i;
        float x2, y;
        const float threehalfs = 1.5F;

        x2 = number * 0.5F;
        y  = number;
        i  = * ( long * ) &y;                       // evil floating point
        ↪  bit level hacking
        i  = 0x5f3759df - ( i >> 1 );               // what the f*ck?
        y  = * ( float * ) &i;
        y  = y * ( threehalfs - ( x2 * y * y ) );   // 1st iteration
//      y  = y * ( threehalfs - ( x2 * y * y ) );   // 2nd iteration, this
↪   can be removed

        return y;
}
```

Listing 4.97. Obscured Intent Code Smell (Quake 3 Arena: Fast Inverse Square Root Example)

4.9.6. Status Variables

Origin: Marcel Jerzyk, "Code Smells: A Comprehensive Online Catalog and Taxonomy" (2022)
Obstruction: Obfuscators
Expanse: Within a Class
Occurrence: Unnecessary Complexity
Hierarchies: Code Smell, Implementation Smell

Status Variables are mutable primitives that are initialized before an operation to store some information based on the process 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 most likely exists a proper solution using first-class functions. Sometimes, they clutter the code, demanding other methods or classes to make additional checks [Section 4.2.7] before execution resulting in Required Setup/Teardown Code [Section 4.1.9].

Causation:

The developer might have special cases that could be handled only inside a loop and could not figure out a better solution.

Problems:

1. Comprehensibility
   - It is more difficult to understand the inner workings of a method than the declarative solution.

Refactor Methods:

1. Replace with Built-In,
2. Extract Method,
3. Remove Status Variables.

```python
def find_foo_index(names: list[str]):
    found = False
    i = 0
    while not found:
        if names[i] == 'foo':
            found = True
        else:
            i += 1
    return i
```

Listing 4.98. Status Variable Code Smell

```python
# Better, but Clever Code
def find_foo_index(names: list[str]):
    for index, value in enumerate(names):
        if value == 'foo':
            return index

# Best
def find_foo_index(names: list[str]):
    return names.index('foo')
```

Listing 4.99. Status Variable Fix

4.9.7. Vertical Separation

Known As: Regions
Origin: Robert Cecil Martin, "Clean Code: A Handbook of Agile Software Craftsmanship" (2008) [55]
Obstruction: Obfuscator
Expanse: Between Classes
Occurrence: Unnecessary Complexity
Hierarchies: Code Smell

There is a tendency to declare variables in one place together before the main "logic" of the method begins. This detachment creates an artificial vertical separation between the variables and the place where they are used. This distancing is an undesirable situation. The same applies to private, utility, or helper functions, which developers should preferably find directly under their first usage [55].

There are programming languages (like C#) that support *Regions* as shown in Listing 4.100, which is a code smell that regards the very same issue, but also supports it through the offered functionality. There are also IDE and editor add-ons [Listing 4.101] that add this externally as a feature. Regions are markers that allow the code to collapse from one particular place to another. It may seem innocent, although this is just a plaster for *Vertical Separation* 's unhealed wound. It is often used to conceal that a method or class is too large, which does not solve the underlying problem.

Sometimes it is used, regardless of the size of the code, to mark separate "regions" for fields, properties, public methods, private methods - even if there are none implemented, to have a certain common standard.

If the code is well written, the default collapsible places based on the code (on methods or classes) should be good enough as an organizing tool.

Causation:

The Vertical Separation may result from the past optimization habits, which were required a long time ago. However, nowadays, the code compilers will optimize it (function variables are put on the stack before the first line of code even gets executed). This could also be a matter of personal preference, but the standard rule is that things should not be too far from each other.

Regions could be used to hide bloat and be a very inexpensive deodorant for other code smells such as Clever Code [Section 4.9.1], Long Method [Section 4.1.4], Imperative Loops [Section 4.7.1], but it just bloats the code even further.

Problems:

1. Encouraged Grouping by Visibility instead of Functionality
   - It is much better to group code by functionality. It leads to better cohesion.
2. Hidden Perspective

    - To see the code in which the regions are used, more clicks are required due to unfolding.
3. Law of Demeter Principle Violation
    - Things related to each other should be as close to each other as possible.
4. No Additional Value

Refactor Methods:

1. Remove the Code Smells instead of Hiding Them.

```
public class Foo
{
  #region Constructor
  public Foo() {}
  #endregion
  #region Methods
  ...
  #endregion
}
```

Listing 4.100. Vertical Separation (Regions) Code Smell: C# Regions

```
class Foo:
    def __init__(self, args):
        #region InitVariables
        ...
        #endregion
```

Listing 4.101. Vertical Separation (Regions) Code Smell: Python Add-On

```
repeat = 5

...

doing_something()
doing_something_else()

...

for index in range(0, repeat):
    ...
```

Listing 4.102. Vertical Separation Code Smell: Distanced Declaration-Usage

```
...

doing_something()
doing_something_else()

...


repeat = 5
for index in range(0, repeat):
    ...
```

Listing 4.103. Vertical Separation Fix: Usage right after Declaration

4.10. OTHERS

This last category contains only one element, which does not fit to the other ones. Existence of that smell is conditioned by external factors.

4.10.1. Incomplete Library Class

Origin: Fowler M., Kent B., "Refactoring: Improving the design of existing code" (1999) [56]
Obstruction: Other
Expanse: Between Classes
Occurrence: Interfaces
Hierarchies: Code Smell

Libraries are the savior of time and money in the industry of software development. If we had to reimplement our algorithms and tools every time we want to use them, then the world around us would not look the same as the world we can now see through the windows. Sometimes, however, a specific package is almost perfect for our needs but lacks some of the functionalities we want. We can, of course, ask the authors whether they plan or add what is missing, or we will try to implement what is missing if the community of a given repository allows it. Unfortunately, sometimes it is impossible, and such situations should also be dealt with by building an additional layer or introducing a foreign method. In the worst case, i.e., reimplementation, we doom the code to *Duplication* [Section 4.4.2].

Causation:

Often open-licensed library makers do this pro bono, so they cannot spend all of their time creating the perfect package for everyone to use.

Problems:

1. Duplication
   - If it's not possible to reuse code that has been already written, then reimplementing it from scratch is massive duplication of work.

Refactor Methods:

1. Introduce Foreign Method,
2. Introduce Local Extension.

5. CODE SMELLS DISCUSSION

The generalization of an existing concept is not new. That is what Fowler did with the *Switch Statement* and *Lazy Class*, renaming them *Repeated Switches* and *Lazy Element*, respectively. One of the reasons for this change was that after the publication of his book, back in 1999, which turned out to be quite revolutionary, the `if/else` and `switch/case` statements gained too many bad *PR*, exchanging the reputation points in favor of polymorphism. The *Lazy Element*, on the other hand, was given a new name simply because of the generalization of the concept, broader than just the classes themselves. Finally, there was also a change from *Inappropriate Intimacy* to *Insider Trading* which is also about the generalization of the described concept, but also an aesthetic change to a more exciting name (probably to make people notice the change).

To sum up my research, the two existing and well-established Code Smells are the most controversial. These controversies stem mainly from the overly general name chosen for them, which includes concepts that are not code smell at all. This creates a willingness to address the problem and an immediate response that "it depends" when asked if it is a code smell. This "it depends" can be treated as some inaccuracy in the naming of the method that results in unwanted "`ifs`". Going back to the Fowler definition, these "`ifs`" are acceptable as code smell is only meant to indicate undesirable quality, not a certainty or a guarantee of it. Nevertheless, it is worth creating new terms that will better capture the set of unpleasant concepts, leaving alone the current subset that does not raise quality objections.

5.1. ISSUE WITH COMMENTS

Regardless of the developer's experience, there certainly are situations where comments are not only helpful but mandatory to give any clue to the reader. For example, when the code is written by developers that have special domain knowledge on some subject (i.e., radio waves, particle behavior). In fact, any comments explaining *why* something is written in a given way could be highly appreciated by other regular developers, who are not expected to understand the specifics of the domain technology but to comprehend the code regardless. Developers have a variety of environment backgrounds, and some code that is readable for one subgroup can be challenging to follow for others.

In extreme cases, when working on very niche, novel, and/or complicated algorithms, the code should usually have explanations bound to them. There is a famous saying of the mathematician who made the ϵ (epsilon) and δ (delta) definitions:

When I wrote this, only God and I understood what I was doing.
Now, God only knows.
Karl Weierstrass

Sometimes, the code is written in a way that is not obvious to others and has a valid reason why things were written in a specific way. This is another example in which a code comment explaining why these decisions were taken would be helpful. The need for such an explanation can occur for a variety of reasons. This could be pre-planned and premeditated because of some specific technical integration of the whole system. A non-obvious design was implemented, which might seem obscure from the code point of view without a broader perspective. The need for this type of comment may also come naturally during the code review process. It may result directly from the observation of the other interested developers, and, after appropriate discussions (if the reasons to do things in a certain way turn out to have a valid point), it may contain explanations to the questions that arose.

There are also optimization cases where the code was intentionally written to maximize the performance sacrificing the readability. These code blocks should preferably contain some comments on *why* and *how* this was achieved.

However, in general, these few specific cases constitute a rather decisive minority of comments that may appear inside the code. It is due to the overwhelming majority of other cases, that the code smell *Comments* is named as is. Nevertheless, it discriminates against the handy use cases mentioned above. Coming back to Fowler's definition of code smell, a given code smell *may* (not *must*) indicate a suspicious situation in the system, so one could argue that the definition and naming are correct. So what are these majority cases?

Let me examine a few of them, which I will recall as a new hierarchy of bad smells, *Comment Smells*. Starting with *"Group Label"* Comment Smell presented in Listing 5.1, which could be easily transformed into a new method to convey the meaning through the method's signature. The same goes for *"Tag"* Comment Smell, where comments compensate for the lack of meaning of variables (or method/class names), as seen in Listing 5.2. When you think of these two, both exist only because of the different code smells they try to compensate for: *Long Method* [Section 4.1.4] and *Uncommunicative Name* [Section 4.8.6] respectively. Once they are fixed, the smelly comments should be removed along with them. If they were not removed, then we would encounter a *"Redundant"* Comment Smell (Listing 5.3).

Another causation of a *"Redundant"* Comment Smell could happen due to enforced global docstring rule as in Listing 5.4 - code could be already in a perfect self-explanatory state and the comment is just cluttering space by duplication. Last but not least, we have a *"Rectify"* Comment Smell, which tries to compensate for a weak attempt of naming something reasonably well, as in Listing 5.5.

```
# Smelly
class Foo:

    def goo(...):
        ...
        # Creating Report
        vanilla_report = get_vanilla_report(...)
        tweaked_report = tweaking_report(vanilla_report, ...)
        final_report = format_report(tweaked_report, ...)
        ...

# Refactored
class Foo:

    def goo(...):
        ...
        report = self.create_report(...)
        ...

    def create_report(self, ...):
        vanilla_report = get_vanilla_report(...)
        tweaked_report = tweaking_report(vanilla_report, ...)
        return format_report(tweaked_report, ...)
```

Listing 5.1. Example of "Group Label" Comment Code Smell

5.1.1. New Proposition: "What" Comments

In conclusion of the above, if a developer is asked if *Comments* are code smells, the correct answer must be "*It depends.*". Why not remove this divagation and make the term more precise, so it is no longer controversial? I would stand for *"What" Comments* because it preserves the "spot chance" for the vast majority of code smells. On the contrary, *"Why" Comments* fulfill their niche and can say what would be challenging to contain within the executable code elegantly. *"What" Comments* as a title for a code smell is informative enough, giving a hint that the code could have been written better instead of making up for it with a built-in explanation. This clear distinction should be much less controversial and significantly improve the meaning by this simple addition of the attributive added to its name.

```python
# Smelly
def get_gross_value(p, t):  # price, tax
    ...

# Refactored
def get_gross_value(price, tax):
    ...
```

Listing 5.2. Example of "Tag" Comment Smell

```python
# Smelly
def increase_attack(self, value: int):
    """
    Increases the attack by the given value.
    """
    ...

# Valid Docstring Case
def destroy_character(character_id: int):
    """
    Removes the character safely from the main game world scene
    if it exists. Safely means it will just throw a warning if
    a character was already removed. The character could be removed due
    to some other trigger event (further info, reference MR#2439).
    """
    ...
```

Listing 5.3. Example of "Redundant" Comment Smell (case 1.)

```python
# Smelly
def open_json_file(filename: str) -> dict:
    """
    Opens file in json format.
    :: params
    - filename (str): name of the file

    :: returns
    - file json (dict)
    """
    ...

# Refactored
def open_json_file(filename: str) -> dict:
    ...
```

Listing 5.4. Example of "Redundant" Comment Smell (case 2.)

```python
# Smelly
def rename_description(product, manufacturer):
    """
    Renames the product description by adding
    the manufacturer footer to the product description.
    """
    ...

# Refactored
def add_manufacturer_footer_to_product_description(product, manufacturer):
    ...
```

Listing 5.5. Example of "Rectify" Comment Smell

The "`ifs`" to the descriptions of the *Comment* code smell were added by Fowler and Wake themselves. Fowler, in the very first sentence at the time of defining it, wrote, *"Do not worry, we are not saying that people should not write comments"*, saying right after that it is just used too often as a "deodorant" for other smells and keeps his statement unchanged in his newest book. Wake explicitly noted that the *"Why" Comments*, and comments explaining non-obvious algorithms, are constructive. I think it is worth distinguishing between these types of comments by title.

5.1.2. New Proposition: Fallacious Comment

There is one extreme case of comment smell that has its exclusive name. It could initially have been an element of smelly *"What" Comments* and reasonable *"Why" Comments*, although this may often be due to the duplication caused by *Redundant* Comment Smell.

If no one has bothered to refactor these comments over time, they may also expire and contain outdated information. It is not something spectacularly unlikely to happen as comments are not part of the token list, which the lexer needs to pass to a parser and examine/execute deliberately. This lack of execution parsing is why a developer or automatic test could accidentally omit it and thus making it even worse than the *Redundant* Comment Smell or than no *Comment* at all - it just would be confusing.

5.2. ISSUE WITH DATA CLASSES

In Object-Oriented Programming, the functionality should be packaged along with its data. It is considered not only a good practice but a principle [38]. This is why a class consisting only of getters and setters could be considered smelly. However, similarly to the *"Comments"* discussion, it does mean that every single case of a class-like construct responsible for holding the data is inherently wrong. There are cases where such a class has many significant advantages over the alternatives, and a developer should not be afraid of using it.

One of the reasons for that is that data classes are a quick tool to combat two other indisputably smelly code smells: *Data Clumps* [Section 4.1.2] and *Primitive Obsession* [Section 4.1.7]. Regarding the Data Clumps, even though adding another class to the system might increase the overall complexity, it is not without benefits. First, it might reduce internal complexity by packing up variables (removing *Data Clumps*) to a meaningful abstracted construct, thus evening out on its cost. However, more than that, one may also gain an informative value if the data class gets a proper name for what it holds. This encapsulation is a solid alternative to having a bunch of separately "floating" variables.

Consider the following Listing 5.6 in which suppose that we cannot control the output of the `get_last_commit()` function because it is given as is from an imported package or an API call. In *Method 1*, the code had to be unpacked into a list of variables and then passed one by one through the `foo` method. Instead, in *Method 2*, the return of `get_last_commit()` is immediately unpacked in an expected container (data class) to store all values within the `Commit` data class. The `foo` method requires only one parameter (thus removing *Long Parameter List*), and the code has been cut in length. This benefits readability because a developer can see that the return is expected to be of the type `Commit` with all the expected fields. Furthermore, in the case of using those data in a place far enough away to make them ambiguous without the receiving context, the collected data will still be encapsulated in the `Commit` class, so there will be less of a problem with the comprehensibility of what these variables are (`commit.files` as opposed to `files`).

Regarding *Primitive Obsession*, the structure of the data class is better than a primitive variable because it abstracts the information instead of having it embedded in the name of the variable. Moreover, these primitives have closed the door to extendability. If ever a need for any additional processing was required, it would not be possible. I illustrate this with the date format Listing 5.7, in which changing the month would not be a trivial task, unless it is encapsulated in a data class.

Another essential benefit of data classes cannot be omitted in any typed language that I did not mention in the `get_last_commit()` example. They can be a scheme for some expected inputs with the benefit of type verification. With any API call, configuration load, or JSON read, data can be directly parsed into a data class object rather than kept as a possible `dict[any]`-type response. Humans make errors, and packages are updated; the constantly changing environment requires standardization. If a developer knows what input he shall expect from the `config.json`, he could pass it directly to a class instead of keeping it in a long dictionary. If a field is missing or has a wrong type, it is detected right after the initialization process of a new object (following the Fail-Fast principle).

Data classes are a fantastic utility to deal with these kinds of problems, and developers should not be afraid to use them where they see the possibility. Of course, after its implementation, it is wise to think whether a behavior could be added to the class for further simplification and code quality improvement, but the data class constructs itself is not inherently smelly, as it makes the code open for extension. It is a solution for code smells that are unanimously undesirable and is a much better way of handling data than any other quick alternative because of its meaning.

Now it remains to answer the question, why is it problematic or controversial? With the increase of API's communication between services, due to the modernization of the Internet to the thing that we all know today, the dictionary type currently has its prime time. The communication goes through http requests where *JSON* objects are by far the leading type of information transporter. This is how almost everything today works as a standard.

However, it is not an ideal solution, which began to be noticed, as with subsequent updates and requests to introduce new functionalities, the lengths of these dictionaries grew with them. Large dictionaries are not heavily criticized for some reason, but they are

```python
# Initial State
def foo(author: str, commit_id: str, files: str, sha_id: str, time: str):
    ...

author, commit_id, files, sha_id, time = get_last_commit()
foo(author, commit_id, files, sha_id, time)


# Improved
@dataclass
class Commit:
    author: str
    commit_id: str
    files: List[str]
    sha_id: str
    time: str

def foo(commit: Commit):
    ...

commit = Commit(**get_last_commit())
foo(commit)


# Best Case Scenario
@dataclass
class Commit:
    author: str
    commit_id: str
    files: List[str]
    sha_id: str
    time: str

    def foo(self):
        ...

commit = Commit(**get_last_commit())
commit.foo()
```

Listing 5.6. The packing of variables in a Data Class in Python

```
# Method 1.
# adjusting the birthday month value is not trivial
birthday_date: str = "1998-03-04"
release_date: str = "2021-03-20"


# Method 2.
@dataclass
class Date
    year: int
    month: int
    day: int

    def __str__(self):
        return f"{self.year}-{self.month}-{self.day}"

birthday: Date = Date(1998, 03, 04)
release: Date = Date(2021, 03, 20)
```

Listing 5.7. Primitive Obsession Refactored into Data Class in Python

just a worse alternative to *Data Class*. They are just a bunch of keys with their respective values (violating *Primitive Obsession*) but without the inspection properties that a simple data class could provide in modern-day editors or IDEs. Not only that, they cannot be adequately annotated at the field level. This could be a completely new code smell named *"Passing Long Dictionary"* due to the critical problems it causes.

Please, note that I do not try to convey that this is a lousy format for communicating information - it is the most popular DTO[1] among developers for a reason. The smell begins to be felt only when the object is not parsed into a proper class, interface, or at least a data class object to hold its properties. This serialization ensures proper typing (which is used extensively in current editors) and solves the problem of ambiguity of the keys of a potential incoming dictionary.

This issue is also addressed by an alternative form of external cross-communication, GraphQL[2]. This tool was created to optimize the amount of data processed and the amount of data transferred by the possibility of selecting the particular values that one would like to request, specifically to deal with the smell *"Passing Long Dictionary"* mentioned above.

These large dictionaries can not only arrive from some external destinations. Enormous local configuration files could have also caused them. There is nothing wrong with that, but only if a proper class or data class instantly encodes these "config.json-s". It would

[1] DTO - Data Transfer Object
[2] GraphQL Website - https://graphql.org/

be a great pity if in a system in which, due to the lack of better alternatives (because of external constraints like available time and effort), the possibility of creating a data class was noticed but not used, and the configuration was left in the dictionary format, just because of the titular name of a code smell. There is also a tool that tries to remedy this localized version of the problem, *JSON Schema*[3] - it provides the annotation and validation for *JSON* documents.

5.2.1. New Proposition: Fate over Action

To preserve the current idea behind the *Data Class* code smell, I propose a new one that could take its place: *Fate over Action*. This Code Smell would signify that the problem is not with the data class concept itself but instead with a situation where external classes or functions primarily manipulate the fields of an object.

The term *Fate over Action* is inspired by personality psychology, precisely from the *Locus of Control* subject. Locus of control is the degree to which a man believes that he has control over the outcome of events in his life. The other end of the spectrum claims that things are beyond one's influence. It could be internal (a belief that one has control over his life) or external (a belief that the outside controls life; by things he cannot influence; be that a chance or *fate*). The Object-Oriented Programming principles gravitate strongly toward a strong internal locus of control - classes should take care of their actions.

This change would remove the stigma of the *Data Class* and pass it on to the term *Fate over Action*, which captures the essence of the problem. The term is a loose proposition; some other candidates for the name could be *Fate over Internal Locus* or just *External Locus of Control*, but I think the one I proposed is the most "catchy" and preserves the idea behind the smell. This name should be less controversial than the current *Data Class*.

[3] JSON Schema Website - `https://json-schema.org/`

6. CODE SMELLS CATALOG

The catalog as a data source 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. I hope that the very simplified form of adding and correcting information will mitigate any problems addressed in the final settlement of this contribution.

6.1. INFORMATIVE CONTENT

I decided to address the potential recursive emergence of the code smell "information scattering" problem by creating a cataloging tool. The goal behind the development of this tool was the simplicity of updating and adding new information. All the essential information can be found in the main `content` directory. Inside this directory, there are Markdown files, where each separate file contains information about one particular Code Smell. All collectible data are in the pinch of the file, in its header, which is structured using the `.yaml` [1] format, presented in Listing 6.1 and Listing 6.2.

Among the currently inserted data, there are synonyms "`known_as`", suggestions for refactoring, "`refactors`", problems that the smell may cause, or principles it violates "`problems`". There is also categorization information, including the groupings mentioned before: "`occurrence`" in Section 3.1.3, "`expanse`" in Section 3.1.2 and "`obstruction`" in Section 3.1.1. There is also a field that shows the possible perspective from which the smell can be considered (the hierarchy): "`type_of_smell`" as discussed in Section 3.2. Additionally, historical information was included attributing the first appearance in the literature, including its type, title, author, and year. This header may be a standardized format for information, but it is only part of what is in the files.

The majority of the data inside the file is informative content right below the header. There, I tried to present the characteristics of the Code Smell, quasi-defining and explaining it in a general and accessible way. There are my conclusive observations based on the collected information, but also the references to the data found in the literature. For the vast majority of issues, I have created minimal code examples that illustrate what a given code smell may look like and also its potential solution.

For the convenience of data research, all those data can be extracted in `.json` format using the Python script inside the repository prepared for that particular task. To run it, the

[1] YAML - `https://yaml.org/`

only requirement is Python in version `3.8+` and one Python package - instructions can be found inside the repository.

6.2. TECHNICALITIES

6.2.1. Implementation

The catalog is a static page automatically built based on the files inside the "content" folder. After contributing changes to the repository, the page will be built automatically using the predefined pipeline process for GitHub Actions, which eliminates the need for any knowledge about page building or deployment in order to be able to contribute from the substantive side.

The mechanism to retrieve data from a local folder is possible thanks to the Gatsby framework [2] and GraphQL [3]. The page was implemented using the React framework [4] with Material UI [5] components, written mainly in TypeScript [6] language. The choice of technology was based on the popularity and modernity of solutions so that the website can have many opportunities for community support and to age visually as slowly as possible.

6.2.2. Platform

There is one more thing worth mentioning. The website's source code is hosted on the GitHub platform, which can be used as a discussion platform. New discussions can be conducted using the *Issues* functionality. This will allow for the exchange of information and insights and clarification of misunderstandings or ambiguity of interpretations. The controversy can be directly addressed and discussed, which will allow a more precise determination of whether a given issue indicates a code of questionable quality.

[2] Gatsby - `https://www.gatsbyjs.com/`
[3] GraphQL - `https://graphql.org/`
[4] React - `https://reactjs.org/`
[5] Material UI for React - `https://mui.com/`
[6] React - `https://www.typescriptlang.org/`

```yaml
slug: "complicated-boolean-expression"
meta:
  last_update_date: 2022-02-20
  title: "Complicated Boolean Expression"
  known_as:
    - ---
categories:
  expanse: "Within"
  obstruction: "Obfuscators"
  occurrence:
    - Conditional Logic
  tags:
    - ---
  type_of_smell:
    - Code Smell
relations:
  related_smells:
    - name: Complicated Regex Expression
      slug: complicated-regex-expression
      type:
        - family
    - name: Obscured Intention
      slug: obscured-intention
      type:
        - causes
    - name: Flag Argument
      slug: flag-arguments
      type:
        - caused
    - name: '"What" Comments'
      slug: what-comments
      type:
        - causes
```

Listing 6.1. Header Example (for Complicated Boolean Expression) [part 1/2]

```
problems:
  general:
    - Comprehensibility
  violation:
    principles:
      - ---
    patterns:
      - ---
refactors:
  - Introduce Explaining Method or Variable
  - Use Guard Clauses
  - Simplify Conditional
history:
  - author: "William C. Wake"
    type: "origin"
    named_as:
      - Complicated Boolean Expression
    regarded_as:
      - Code Smell
    source:
      year: 2004
      authors:
        - William C. Wake
      name: "Refactoring Workbook"
      type: "book"
      href:
        isbn_13: "978-0321109293"
        isbn_10: "0321109295"
```

Listing 6.2. Header Example (for Complicated Boolean Expression) [part 2/2]

CONCLUSION

This last section is the culmination of this Master's Thesis. In it, I remind the reader that it should be treated with an appropriate handful of salt, because the breadth of the topic leaves a broad field in which errors and shortcomings could appear. I also indicate a few potential future paths for the development of the subject of this thesis as well as the most important added value that this thesis presents.

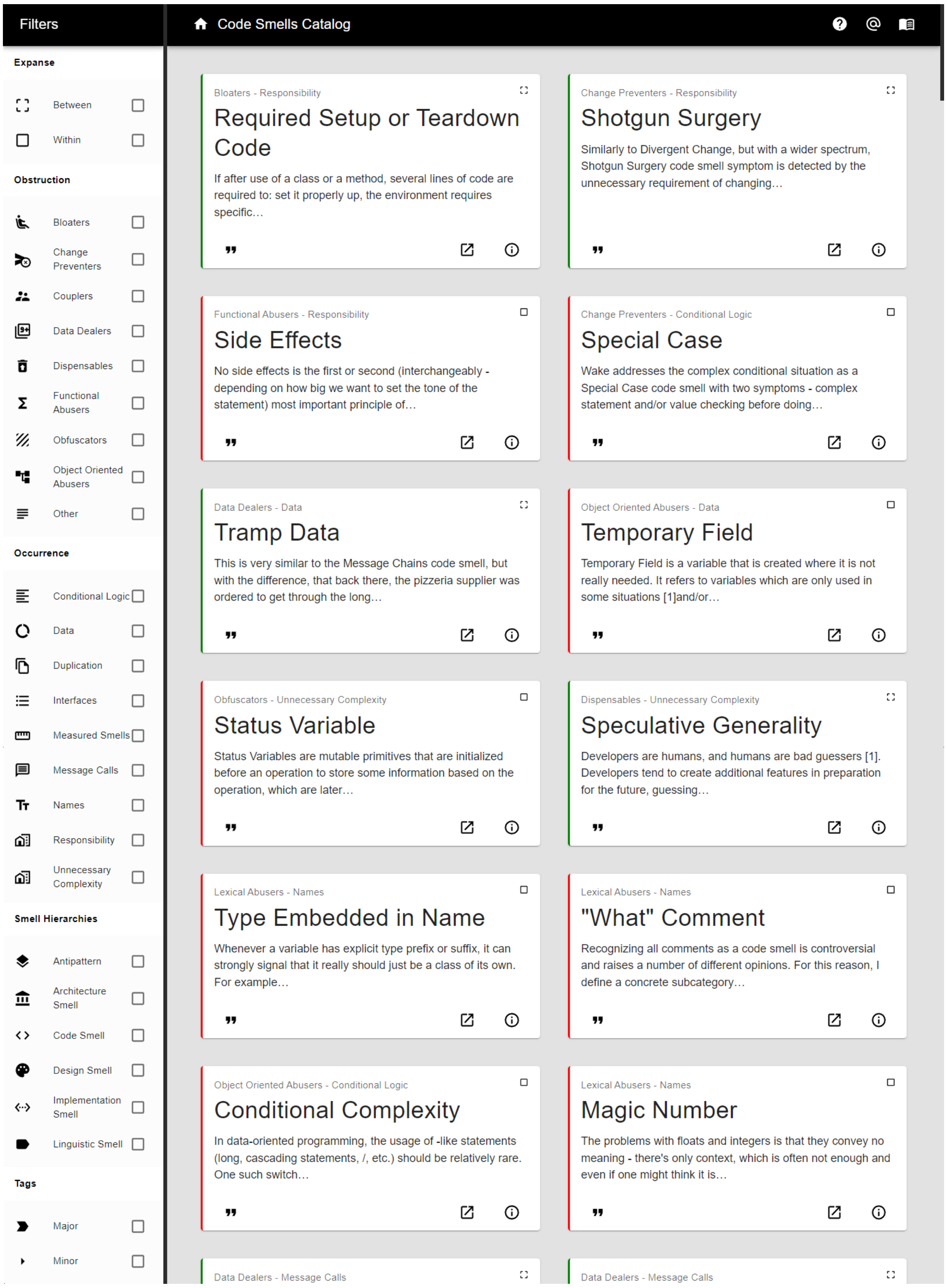
ACHIEVED GOALS

The main goal of the work, which was to create an open catalog, was successfully achieved. I also met additional goals about the simplicity of contributing new knowledge to the catalog and the other assumption that the catalog should be a great source of information for potential new people willing to delve into the topic and a tool for immediate data extraction for researchers. It was made using modern technologies with a non-deterrent graphic design, as shown in the Figure 6.1 so as not to discourage visitors from getting to know the topic more in-depth. As for the extraction - the data is kept in the headers, from which the researcher can generate more universal-to-use data files in JSON format with the scripts provided in the repository.

The catalog database is currently the most extensive and comprehensive source of information about the Code Smells. All the existing Code Smells from the literature were collected, and I extracted new ones based on the grey data and the author's domain knowledge. There are examples and solutions for each of the Code Smells, which are in-existent in the currently available information and, thus - unique. Values and taxonomies as attributes have been collected there and assigned, which may open the way to systematic and standardized research in the developing area of Code Smells from the point of view of machine learning.

In addition, I drew attention to the blurring concepts in the field of Bad Smells, along with a proposal for a new concept of *hierarchies* along with the collection of existing issues of this type in the form of a list.

Along with this Master's Thesis, a research paper article was also created and submitted to Springer so that this work would not go unnoticed, as it conveys essential value in the field of code quality.



Filters

Expanse

- Between
- Within

Obstruction

- Bloaters
- Change Preventers
- Couplers
- Data Dealers
- Dispensables
- Functional Abusers
- Obfuscators
- Object Oriented Abusers
- Other

Occurrence

- Conditional Logic
- Data
- Duplication
- Interfaces
- Measured Smells
- Message Calls
- Names
- Responsibility
- Unnecessary Complexity

Smell Hierarchies

- Antipattern
- Architecture Smell
- Code Smell
- Design Smell
- Implementation Smell
- Linguistic Smell

Tags

- Major
- Minor

Code Smells Catalog

Bloaters - Responsibility

Required Setup or Teardown Code

If after use of a class or a method, several lines of code are required to: set it properly up, the environment requires specific…

Change Preventers - Responsibility

Shotgun Surgery

Similarly to Divergent Change, but with a wider spectrum, Shotgun Surgery code smell symptom is detected by the unnecessary requirement of changing…

Functional Abusers - Responsibility

Side Effects

No side effects is the first or second (interchangeably - depending on how big we want to set the tone of the statement) most important principle of…

Change Preventers - Conditional Logic

Special Case

Wake addresses the complex conditional situation as a Special Case code smell with two symptoms - complex statement and/or value checking before doing…

Data Dealers - Data

Tramp Data

This is very similar to the Message Chains code smell, but with the difference, that back there, the pizzeria supplier was ordered to get through the long…

Object Oriented Abusers - Data

Temporary Field

Temporary Field is a variable that is created where it is not really needed. It refers to variables which are only used in some situations [1]and/or…

Obfuscators - Unnecessary Complexity

Status Variable

Status Variables are mutable primitives that are initialized before an operation to store some information based on the operation, which are later…

Dispensables - Unnecessary Complexity

Speculative Generality

Developers are humans, and humans are bad guessers [1]. Developers tend to create additional features in preparation for the future, guessing…

Lexical Abusers - Names

Type Embedded in Name

Whenever a variable has explicit type prefix or suffix, it can strongly signal that it really should just be a class of its own. For example…

Lexical Abusers - Names

"What" Comment

Recognizing all comments as a code smell is controversial and raises a number of different opinions. For this reason, I define a concrete subcategory…

Object Oriented Abusers - Conditional Logic

Conditional Complexity

In data-oriented programming, the usage of -like statements (long, cascading statements, /, etc.) should be relatively rare. One such switch…

Lexical Abusers - Names

Magic Number

The problems with floats and integers is that they convey no meaning - there's only context, which is often not enough and even if one might think it is…

Data Dealers - Message Calls

Data Dealers - Message Calls

Fig. 6.1. Code Smell Catalog - `https://luzkan.github.io/smells/`

LIMITATIONS & THREATS TO VALIDITY

I tried to contribute with the best intentions I could in both this thesis and the *Code Smells Catalog* attached to it, but despite my best efforts, I believe, just from the statistical point of view, the field is so substantial that I must be wrong in some cases. I might have overlooked or twisted some information, or in cases where there were no more credible sources, and I had to rely on my expertise and expertise of my colleagues from the industry and university - I could have come up to conclusions that were not flawless (e.g., mistakenly assigning an attribute or category). I also considered addressing all smells that are considered quite resonantly controversial, which **is** controversial, and it could not be any other way. I came up with such terms, which after the title itself, were not contentious and such that they would miss conveying through their title ideas that are not intrinsically wrong. This action should raise some debate as to whether such changes were reasonable.

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

FURTHER RESEARCH

There are also matters that I cannot address myself. While compiling the list of all the current Code Smells, I focused on what Mäntylä concentrated on when creating its taxonomy in 2003: selecting and creating names that will bring the most effective understanding of the topic while being just granular enough. One can say that it is a particular optimization issue, and we know that the computer or a wider group of people can do it much better. Did I do something wrong by collapsing all the *Large Class* related smells into one? I did not want them to confuse the reader. Still, maybe there is a possibility to go back to all these synonyms, research whether it is not possible to extract maybe one (or two) more different smell(-s) from them, which is just distinct enough (and explicitly write, what is the difference between them, or maybe why is he such a characteristic "son" or "parent" of a given smell to be distinct)? Perhaps, in general, the synonymous names in some of the Code Smells should be more granular and have their pedestal for display? Maybe the fragmentation should only occur when viewed at the right angle, e.g., when "sniffing" the smells from the code perspective; regarding abstraction problems, we would have just the three smells that I have listed out, but from the design perspective, there could be much more and specific smells? There is a need to check whether my thoughts are valid, make the definitions or constraints, and verify whether it is currently all right or, if not, change it to make it right.

Another thing that I drew attention to in the problems section is the concepts of the smell hierarchies. Finally, I assumed that individual smells are not in a bijection relationship with smell categories. In other words, a given smell can be both a lexical smell, code smell, or a design smell at the same time. Whether this is correct (or desirable) should also be considered. Ideally, this should be done when the framework and definitions of each smell category are clearly and unambiguously defined.

I would address the essential thing - **comprehensibility**, as it turned out to be universal over time and the desired value (by giving a counterexample, reusability turned out to be controversial because although good in assumption, it rarely turns out to be useful in practice in larger contexts). Following the thought of comprehension, I would define the categories of smells by intuitively perceiving what Fowler did in 1999 - something is wrong here, expanding on that by understanding *"here"* in the context of what we are perceiving (looking at): *code*, *design*, *architecture*, *spelling*, etc. Taking such a position implies that the bijection relationship between the category and smell cannot exist - looking at the code, I can see that something is wrong with the design - e.g., Insider Trading takes place. Thus I could refer to it both as *Code Smell* and *Design Smell*. The second option to consider, and often repeated in the definitions for various smell hierarchies, is "a collection of bad practices", but this definition is the core of the problem that makes the antipatterns a too-similar concept. *Bad smell* and *antipattern* would refer to the same thing and could be used interchangeably as synonyms if that was the case. However, is it desirable? There is a difference in these concepts, and it would be worth keeping and distinguishing them, but this may be a moot point. My proposal to distinguish between these concepts should be reviewed. Taking this opportunity to appeal, I think it would be beneficial to name the individual categories of Smells just by the name of their category and not to mention them synonymously with the name *Bad Smells*. Then, the phrase Bad Smells could be the umbrella term (global term) for all categories and smells, eliminating any confusing terminology and phrases in existing usage, cementing the correctness of the research carried out so far.

SUMMARY

This is a ready, open-source database about Code Smells, which is not only a spiritless source but also serves as an informative web application for everyone. This directly addresses the lack of awareness among developers, which has been shown to impact the quality of the software produced. However, beyond the availability of information for all programmers, it is meant to be used for further research. There is a script that transforms data into `.json` files so that, if anyone wants, they can extract the current data with a single command, grind something with an algorithm, and contribute further in the field of software quality. This will allow us to collect consistent results without accidentally (even in the absence of awareness) bypassing a significant amount of Code Smells, as was previously. 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. In addition to the catalog itself, I hope this work will broaden readers' awareness of the different Bad Smells hierarchies and allow more precise use of their specific terms, including Bad Smell and Antipattern.

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.

LIST OF FIGURES

LIST OF LISTINGS

LIST OF TABLES

BIBLIOGRAPHY

[1] Al-Shaaby, A., Aljamaan, H., Alshayeb, M., *Bad Smell Detection Using Machine Learning Techniques: A Systematic Literature Review*, Arabian Journal for Science and Engineering. 2020, tom 45, 4, str. 2341–2369.

[2] Alkharabsheh, K., Crespo, Y., Manso, E., Taboada, J.A., *Software Design Smell Detection: a systematic mapping study*, Software Quality Journal. 2019, tom 27, 3, str. 1069–1148.

[3] Almeida, D., Campos, J.C., Saraiva, J., Silva, J.C., *Towards a catalog of usability smells*, w: *Proceedings of the 30th Annual ACM Symposium on Applied Computing* (2015), str. 175–181.

[4] Arendt, T., Taentzer, G., *UML Model Smells and Model Refactorings in Early Software Development Phases*, Universitat Marburg. 2010.

[5] Arnaoudova, V., Di Penta, M., Antoniol, G., Guéhéneuc, Y.G., *A New Family of Software Anti-patterns: Linguistic Anti-patterns*, w: *2013 17th European Conference on Software Maintenance and Reengineering* (IEEE, 2013), str. 187–196.

[6] Atwood, J., *Code Smells*, Jeff Atwood Website. 2006.
Url (accessed: 03.11.2021):
`https://blog.codinghorror.com/code-smells/`.

[7] Aversano, L., Carpenito, U., Iammarino, M., *An Empirical Study on the Evolution of Design Smells*, Information. 2020, tom 11, 7, str. 348.

[8] Bansiya, J., Davis, C., *A hierarchical model for object-oriented design quality assessment*, IEEE Transactions on Software Engineering. 2002, tom 28, 1, str. 4–17.

[9] Braun, L.T., Borrmann, K.F., Lottspeich, C., Heinrich, D.A., Kiesewetter, J., Fischer, M.R., Schmidmaier, R., *Guessing right – whether and how medical students give incorrect reasons for their correct diagnoses*, GMS Journal for Medical Education. 2019, tom 36, 6.

[10] Buschmann, F., Henney, K., Schmidt, D.C., *Pattern Oriented Software Architecture Volume 5: On Patterns and Pattern Languages*, tom 5 (John wiley & sons, 2007).

[11] Carette, A., Younes, M.A.A., Hecht, G., Moha, N., Rouvoy, R., *Investigating the energy impact of Android smells*, w: *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (IEEE, 2017), str. 115–126.

[12] Chapman, C., Wang, P., Stolee, K.T., *Exploring Regular Expression Comprehension*, w: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)* (IEEE, 2017), str. 405–416.

[13] Cunningham, W., *The WyCash portfolio management system*, ACM SIGPLAN OOPS Messenger. 1992, tom 4, 2, str. 29–30.

[14] D'Ambros, M., Bacchelli, A., Lanza, M., *On the Impact of Design Flaws on Software Defects*, w: *2010 10th International Conference on Quality Software* (IEEE, 2010), str. 23–31.

[15] De Castella, K., Byrne, D., Covington, M., *Unmotivated or Motivated to Fail? A*

Cross-Cultural Study of Achievement Motivation, Fear of Failure, and Student Disengagement, Journal of educational psychology. 2013, tom 105, 3, str. 861.

[16] Diehl, S., *What I Wish I Knew When Learning Haskell*. 2016.
Url (accessed: 15.11.2021):
`http://dev.stephendiehl.com/hask/tutorial_print.pdf`.

[17] Dobbin, F., Kalev, A., *Why Diversity Programs Fail*, Harvard Business Review. 2016, tom 94, 7, str. 14.

[18] Doğan, E., Tüzün, E., *Towards a taxonomy of code review smells*, Information and Software Technology. 2022, tom 142, str. 106737.

[19] Eessaar, E., Kaosaar, E., *On Finding Model Smells Based on Code Smells*, w: *Computer Science On-line Conference* (Springer, 2018), str. 269–281.

[20] Eliazar, I., *Lindy's Law*, Physica A: Statistical Mechanics and its Applications. 2017, tom 486, str. 797–805.

[21] Fokaefs, M., Tsantalis, N., Chatzigeorgiou, A., *JDeodorant: Identification and Removal of Feature Envy Bad Smells*, w: *2007 ieee international conference on software maintenance* (IEEE, 2007), str. 519–520.

[22] Fontana, F.A., Ferme, V., Marino, A., Walter, B., Martenka, P., *Investigating the Impact of Code Smells on System's Quality: An Empirical Study on Systems of Different Application Domains*, w: *2013 IEEE International Conference on Software Maintenance* (IEEE, 2013), str. 260–269.

[23] Fontana, F.A., Lenarduzzi, V., Roveda, R., Taibi, D., *Are architectural smells independent from code smells? An empirical study*, Journal of Systems and Software. 2019, tom 154, str. 139–156.

[24] Fowler, M., *CodeSmell.*, Martin Fowler Website. 2006.
Url (accessed: 09.11.2021):
`https://martinfowler.com/bliki/CodeSmell.html`.

[25] Fowler, M., *Flag Argument*, Martin Fowler Website. 2006.
Url (accessed: 16.11.2021):
`https://martinfowler.com/bliki/FlagArgument.html`.

[26] Fowler, M., *Refactoring: Improving the Design of Existing Code* (Addison-Wesley Professional, 2018).

[27] Ganesh, S., Sharma, T., Suryanarayana, G., *Towards a Principle-based Classification of Structural Design Smells*, J. Object Technol. 2013, tom 12, 2, str. 1–1.

[28] Garcia, J., Popescu, D., Edwards, G., Medvidovic, N., *Identifying Architectural Bad Smells*, w: *2009 13th European Conference on Software Maintenance and Reengineering* (2009), str. 255–258.

[29] Garcia, J., Popescu, D., Edwards, G., Medvidovic, N., *Toward a Catalogue of Architectural Bad Smells*, w: *International conference on the quality of software architectures* (Springer, 2009), str. 146–162.

[30] Ghafari, M., Gadient, P., Nierstrasz, O., *Security Smells in Android*, w: *2017 IEEE 17th*

international working conference on source code analysis and manipulation (SCAM) (IEEE, 2017), str. 121–130.

[31] Graziotin, D., Fagerholm, F., Wang, X., Abrahamsson, P., *Consequences of Unhappiness While Developing Software*, w: *2017 IEEE/ACM 2nd International Workshop on Emotion Awareness in Software Engineering (SEmotion)* (2017), str. 42–47.

[32] Graziotin, D., Wang, X., Abrahamsson, P., *Do feelings matter? On the correlation of affects and the self-assessed productivity in software engineering*, Journal of Software: Evolution and Process. 2015, tom 27, 7, str. 467–487.

[33] Guzman, E., Bruegge, B., *Towards Emotional Awareness in Software Development Teams*, w: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013 (Association for Computing Machinery, New York, NY, USA, 2013), str. 671–674.

[34] Hall, T., Beecham, S., Bowes, D., Gray, D., Counsell, S., *A Systematic Literature Review on Fault Prediction Performance in Software Engineering*, IEEE Transactions on Software Engineering. 2011, tom 38, 6, str. 1276–1304.

[35] Haque, M.S., Carver, J., Atkison, T., *Causes, Impacts, and Detection Approaches of Code Smell: A Survey*, w: *Proceedings of the ACMSE 2018 Conference*, ACMSE '18 (Association for Computing Machinery, New York, NY, USA, 2018), str. 1–8.

[36] Harper, R., *Boolean Blindness*. 2011.
Url (accessed: 15.11.2021):
https://existentialtype.wordpress.com/2011/03/15/boolean-blindness/.

[37] Hermans, F., Pinzger, M., Van Deursen, A., *Detecting and visualizing inter-worksheet smells in spreadsheets*, w: *2012 34th International Conference on Software Engineering (ICSE)* (IEEE, 2012), str. 441–451.

[38] Hunt, A., Thomas, D., *The Pragmatic Programmers* (Addison-Wesley Professional, 1999).

[39] Jabrayilzade, E., Gürkan, O., Tüzün, E., *Towards a taxonomy of inline code comment smells*, w: *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)* (2021), str. 131–135.

[40] Karwin, B., *SQL Antipatterns: Avoiding the Pitfalls of Database Programming* (Pragmatic Bookshelf, 2010).

[41] Kaur, A., *A Systematic Literature Review on Empirical Analysis of the Relationship Between Code Smells and Software Quality Attributes*, Archives of Computational Methods in Engineering. 2020, tom 27, 4, str. 1267–1296.

[42] Kaur, A., Jain, S., Goel, S., Dhiman, G., *Prioritization of code smells in object-oriented software: A review*, Materials Today: Proceedings. 2021.

[43] Kerievsky, J., *Refactoring to patterns* (Pearson Deutschland GmbH, 2005).

[44] Lacerda, G., Petrillo, F., Pimenta, M., Guéhéneuc, Y.G., *Code Smells and Refactoring: A Tertiary Systematic Review of Challenges and Observations*, Journal of Systems and Software. 2020, tom 167, str. 110610.

[45] Lanza Michele, R.M., *Object-Oriented Metrics in Practice: Using Software Metrics to*

Characterize, Evaluate, and Improve the Design of Object-Oriented Systems (Springer-Verlag Berlin Limited, 2007).

[46] Lenarduzzi, V., Sillitti, A., Taibi, D., *A Survey on Code Analysis Tools for Software Maintenance Prediction*, w: *International Conference in Software Engineering for Defence Applications* (Springer, 2018), str. 165–175.

[47] Lewowski, T., Madeyski, L., *How far are we from reproducible research on code smell detection? A systematic literature review*, Information and Software Technology. 2022, tom 144, str. 106783.

[48] Li, W., Shatnawi, R., *An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution*, Journal of systems and software. 2007, tom 80, 7, str. 1120–1128.

[49] Lozano, A., Wermelinger, M., *Assessing the effect of clones on changeability*, w: *2008 IEEE International Conference on Software Maintenance* (IEEE, 2008), str. 227–236.

[50] MacConnell, S., McConnell, S., *Code Complete: A Practical Handbook of Software Construction* (Microsoft Press Redmont, WA, 1993).

[51] Mantyla, M., *Bad Smells in Software - a Taxonomy and an Empirical Study*, Rozprawa doktorska, PhD thesis, Helsinki University of Technology. 2003.
Url (accessed: 11.11.2021):
http://www.soberit.hut.fi/sems/shared/deliverables_public/mmantyla_thesis_final.pdf.

[52] Mantyla, M., Vanhanen, J., Lassenius, C., *A Taxonomy and an Initial Empirical Study of Bad Smells in Code*, w: *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.* (IEEE, 2003), str. 381–384.

[53] Mantyla, M.V., Lassenius, C., *Subjective evaluation of software evolvability using code smells: An empirical study*, Empirical Software Engineering. 2006, tom 11, 3, str. 395–431.

[54] Marticorena, R., López, C., Crespo, Y., *Extending a Taxonomy of Bad Code Smells with Metrics*, w: *Proceedings of 7th International Workshop on Object-Oriented Reengineering (WOOR)* (Citeseer, 2006), str. 6.

[55] Martin, R.C., *Clean Code: A Handbook of Agile Software Craftsmanship* (Pearson Education, 2008).

[56] Martin Fowler, K.B., *"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).

[57] Martini, A., Bosch, J., *The Danger of Architectural Technical Debt: Contagious Debt and Vicious Circles*, w: *2015 12th Working IEEE/IFIP Conference on Software Architecture* (2015), str. 1–10.

[58] McConnell, S., *Code Complete* (Pearson Education, 2004).

[59] Moha, N., Gueheneuc, Y.G., Duchien, L., Le Meur, A.F., *DECOR: A Method for the Specification and Detection of Code and Design Smells*, IEEE Transactions on Software Engineering. 2010, tom 36, 1, str. 20–36.

[60] Moha, N., Gueheneuc, Y.G., Duchien, L., Le Meur, A.F., *DECOR: A Method for the Specification and Detection of Code and Design Smells*, IEEE Transactions on Software Engineering. 2010, tom 36, 1, str. 20–36.

[61] Monden, A., Nakae, D., Kamiya, T., Sato, S., Matsumoto, K., *Software quality analysis by code clones in industrial legacy software*, w: *Proceedings Eighth IEEE Symposium on Software Metrics* (2002), str. 87–94.

[62] Monteiro, M.P., Fernandes, J.M., *Towards a catalog of Aspect-oriented refactorings*, w: *Proceedings of the 4th international conference on Aspect-oriented software development* (2005), str. 111–122.

[63] Mäntylä, M.V., Lassenius, C., *A Taxonomy for "Bad Code Smells"*. 2006. *Url (accessed: 03.11.2021):* `https://web.archive.org/web/20120111101436/http://www.soberit.hut.fi/mmantyla/BadCodeSmellsTaxonomy.htm`.

[64] Müller, S.C., Fritz, T., *Stuck and Frustrated or in Flow and Happy: Sensing Developers' Emotions and Progress*, w: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, tom 1 (2015), str. 688–699.

[65] Olbrich, S., Cruzes, D.S., Basili, V., Zazworka, N., *The evolution and impact of code smells: A case study of two open source systems*, w: *2009 3rd international symposium on empirical software engineering and measurement* (IEEE, 2009), str. 390–400.

[66] Olbrich, S.M., Cruzes, D.S., Sjøberg, D.I., *Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems*, w: *2010 IEEE international conference on software maintenance* (IEEE, 2010), str. 1–10.

[67] Palomba, F., Bavota, G., Penta, M.D., Fasano, F., Oliveto, R., Lucia, A.D., *On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation*, Empirical Software Engineering. 2018, tom 23, 3, str. 1188–1221.

[68] Palomba, F., Tamburri, D.A., Serebrenik, A., Zaidman, A., Arcelli Fontana, F., Oliveto, R., *Poster: How Do Community Smells Influence Code Smells?*, w: *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)* (2018), str. 240–241.

[69] Peters, R., Zaidman, A., *Evaluating the Lifespan of Code Smells using Software Repository Mining*, w: *2012 16th European conference on software maintenance and reengineering* (IEEE, 2012), str. 411–416.

[70] Qamar, K., Sülün, E., Tüzün, E., *Towards a Taxonomy of Bug Tracking Process Smells: A Quantitative Analysis*, w: *2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)* (IEEE, 2021), str. 138–147.

[71] Rowa, K., *Atychiphobia (Fear of Failure)* (ABC-CLIO, 2015).

[72] Sabir, F., Palma, F., Rasool, G., Guéhéneuc, Y.G., Moha, N., *A systematic literature review on the detection of smells and their evolution in object-oriented and service-oriented systems*, Software: Practice and Experience. 2019, tom 49, 1, str. 3–39.

[73] Santos, J.A.M., Rocha-Junior, J.B., Prates, L.C.L., do Nascimento, R.S., Freitas, M.F., de Mendonça, M.G., *A systematic review on the code smell effect*, Journal of Systems and Software. 2018, tom 144, str. 450–477.

[74] Sharma, T., *Presentation smells: How not to prepare your conference presentation*, Tushar Sharma Website. 2016.

Url (accessed: 01.03.2022):

`https://www.tusharma.in/presentation-smells.html`.

[75] Sharma, T., Fragkoulis, M., Spinellis, D., *Does Your Configuration Code Smell?*, w: *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)* (IEEE, 2016), str. 189–200.

[76] Sharma, T., Fragkoulis, M., Spinellis, D., *House of Cards: Code Smells in Open-Source C# Repositories*, w: *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (IEEE, 2017), str. 424–429.

[77] Shatnawi, R., Li, W., *An Investigation of Bad Smells in Object-Oriented Design*, w: *Third International Conference on Information Technology: New Generations (ITNG'06)* (2006), str. 161–165.

[78] Singh, S., Kaur, S., *A systematic literature review: Refactoring for disclosing code smells in object oriented software*, Ain Shams Engineering Journal. 2018, tom 9, 4, str. 2129–2151.

[79] Singjai, A., Simhandl, G., Zdun, U., *On the practitioners' understanding of coupling smells — A grey literature based Grounded-Theory study*, Information and Software Technology. 2021, tom 134, str. 106539.

[80] Sjøberg, D.I., Yamashita, A., Anda, B.C., Mockus, A., Dybå, T., *Quantifying the Effect of Code Smells on Maintenance Effort*, IEEE Transactions on Software Engineering. 2012, tom 39, 8, str. 1144–1156.

[81] Smith, C.U., Williams, L.G., *New Software Performance AntiPatterns: More Ways to Shoot Yourself in the Foot.*, w: *Int. CMG Conference* (Citeseer, 2002), str. 667–674.

[82] Smith, C.U., Williams, L.G., *More New Software Antipatterns: Even More Ways to Shoot Yourself in the Foot.*, w: *Computer Measurement Group Conference* (Citeseer, 2003), str. 717–725.

[83] Smith, S., *Refactoring Fundamentals*. 2013.
Url (accessed: 11.11.2021):

`https://www.pluralsight.com/courses/refactoring-fundamentals`.

[84] Soh, Z., Yamashita, A., Khomh, F., Guéhéneuc, Y.G., *Do Code Smells Impact the Effort of Different Maintenance Programming Activities?*, w: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, tom 1 (IEEE, 2016), str. 393–402.

[85] Stijlaart, M., Zaytsev, V., *Towards a taxonomy of grammar smells*, w: *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering* (2017), str. 43–54.

[86] Suryanarayana, G., Samarthyam, G., Sharma, T., *Refactoring for Software Design Smells: Managing Technical Debt* (Morgan Kaufmann, 2014).

[87] Taylor, R.N., Medvidovic, N., Dashofy, E.M., *Software architecture: foundations, theory, and practice.(2009)*, Google Scholar Google Scholar Digital Library Digital Library. 2009.

[88] Tufano, M., Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., Poshyvanyk, D., *An empirical investigation into the nature of test smells*, w: *Proceedings of the 31st IEEE/ACM international conference on automated software engineering* (2016), str. 4–15.

[89] Van Deursen, A., Moonen, L., Van Den Bergh, A., Kok, G., *Refactoring Test Code*, w: *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)* (Citeseer, 2001), str. 92–95.

[90] Vazou, N., Bakst, A., Jhala, R., *Bounded Refinement Types*, w: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (2015), str. 48–61.

[91] Vetro, A., Ardito, L., Morisio, M., *Definition, implementation and validation of energy code smells: an exploratory study on an embedded system*, None. 2013.

[92] Wake, W.C., *Refactoring Workbook 1st Edition* (Addison-Wesley Professional, 2004).

[93] Yamashita, A., Moonen, L., *Do code smells reflect important maintainability aspects?*, w: *2012 28th IEEE international conference on software maintenance (ICSM)* (IEEE, 2012), str. 306–315.

[94] Yamashita, A., Moonen, L., *Do developers care about code smells? An exploratory survey*, w: *2013 20th working conference on reverse engineering (WCRE)* (IEEE, 2013), str. 242–251.

[95] Yanok, I., Nystrom, N., *Tapir: A Language for Verified OS Kernel Probes*, ACM SIGOPS Operating Systems Review. 2016, tom 49, 2, str. 51–56.

[96] Yli-Huumo, J., Rissanen, T., Maglyas, A., Smolander, K., Sainio, L.M., *The Relationship Between Business Model Experimentation and Technical Debt*, w: *International Conference of Software Business* (Springer, 2015), str. 17–29.

[97] Yu, Z., Rajlich, V., *Hidden dependencies in program comprehension and change propagation*, w: *Proceedings 9th International Workshop on Program Comprehension. IWPC 2001* (IEEE, 2001), str. 293–299.

[98] Zazworka, N., Spínola, R.O., Vetro', A., Shull, F., Seaman, C., *A Case Study on Effectively Identifying Technical Debt*, w: *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*, EASE '13 (Association for Computing Machinery, New York, NY, USA, 2013), str. 42–47.

[99] Zhang, M., Baddoo, N., Wernick, P., Hall, T., *Improving the Precision of Fowler's Definitions of Bad Smells*, w: *2008 32nd Annual IEEE Software Engineering Workshop* (IEEE, 2008), str. 161–166.