Bad Smells in Javascript - A Mapping Study

Aryclenio Xavier Barros
IMD - UFRN
Natal, Brazil
aryclenio.barros.106@ufrn.edu.br

Eiji Adachi
IMD - UFRN
Natal, Brazil
eijiadachi@imd.ufrn.br

ABSTRACT

Javascript is one of the most famous mainstream programming languages nowadays. It has gained considerable practical relevance over the last years, with applications in several areas, such as games, 3D rendering, and, mainly, web development. Like any other software system, systems developed in Javascript need to keep their ability to evolve to remain useful and relevant over time. Empirical evidence has shown that bad smells are possible indicators of problems hindering software evolvability. In this context, this paper presents a mapping study investigating if and to what extent bad smells have been defined for the Javascript language and how the interest in this topic has evolved. Our study identified 26 different types of bad smells investigated in the context of Javascript in 8 different works published between 2013 and 2020. This result suggests that although Javascript has gained practical relevance in recent years, there is still room for further empirical studies defining and evaluating the impact of bad smells on Javascript-based software systems.

KEYWORDS

Javascript, bad smells, code smells, mapping

1 INTRODUCTION

JavaScript, hereafter simply referred to as JS, is a dynamically typed programming language that has gained considerable practical relevance over the last years. Due to its purpose of working with web technologies and the incredible popularity this industry has enjoyed in recent years, the language has become more accessible, cross-functional, widespread, and updated more frequently [11]. According to data from a survey conducted by the Stack Overflow website, the language is the most used among all developers for 5 years in a row. [1]

With the popularization of JS and its growing adoption in the development of web systems, it becomes increasingly important that systems developed with JS technologies are long-lived, i.e., able to evolve to remain useful over the years [18]. But as the system evolves, changes to its source code may deteriorate its evolution ability [4]. Therefore, it is necessary to have the means to early identify problems that deteriorate the evolvability of software systems.

Empirical evidence has pointed to *bad smells* as possible indicators of problems that negatively affect the evolution of software systems [5]. *Bad smells* are defined as bad solutions to problems related to architecture, design, and source code of software applications [9]. The study by Tahir et al. [22], for example, reports that 27% of the problems found in applications are somehow related to *bad smells*.

In this context, this work aims to carry out a mapping study to investigate if and to what extent *bad smells* have already been investigated in the context of Javascript. More specifically, our study aims to answer the following research questions:

- Q1) Which *bad smells* have already been investigated in the context of Javascript?
- Q2) How has the interest in the topic of *bad smells* in Javascript behaved over time?

With these research questions, we explore if and to what extent *bad smells* are being investigated in the context of Javascript and how the interest in this topic has evolved. Our study identified a total of 8 works investigating *bad smells* in the context of JS, from which we extracted 26 different types of *bad smells*. Our results show that the number of works investigating *bad smells* in JS is still quite limited when compared to the number of works considering other programming languages, like Java, and no growing trend is observable. Also, most of the *bad smells* identified in our study are "classic" and generic smells adapted from other works; there are still very few smells defined for features specific to the JS programming language.

The rest of this paper is structured as follows. In Section 2, we present the settings of our mapping study. In Section 3 we present the study of the data obtained to answer the research questions. In Section 4 we analyze the threats to validate our study. Section 5 shows the related works that based our research. And finally, section 6shows our conclusion about the research.

2 SYSTEMATIC MAPPING STUDY

This study aims to carry out a mapping study about *bad smells* in the context of the Javascript programming language to delimit the current state-of-the-art in the literature on this topic. For this, we followed the guidelines defined by Petersen, Vakkalanka, and Kuzniarz for conducting systematic mappings in Software Engineering [17]. In the following sections, we detail our search questions, search string, inclusion and exclusion criteria, and procedure followed to conduct our mapping.

2.1 Research Questions

This paper aims to investigate the research questions below:

- Q1) Which *bad smells* have already been investigated in the context of Javascript?
- Q2) How has the interest in the topic of *bad smells* in Javascript behaved over time?

With the growing popularity of the Javascript language, especially in the development of systems for the Web, and with the relevance of *bad smells* in the context of software maintenance and

[1] *The Stack Overflow Developer Survey 2020.* Available at https://insights.stackoverflow.com/survey/2020

evolution, we sought with these research questions to investigate whether and to what extent studies on *bad smells* in the context of Javascript have been performed in scientific works. Thus, with the first research question, we aim to map the *bad smells* already investigated in the literature, helping to organize the current body of knowledge on this topic. With the second research question, we aim to observe how the interest in the topic has behaved over time, possibly identifying trends in scientific studies that follow the popularity that the Javascript language has gained over the last few years.

2.2 Search String and Data Source

To build the search string for our study, the intersection of the terms Javascript and *bad smell* was used as a basis. For each of these terms, we defined some synonyms to expand the search scope. For the term Javascript, we use as a synonym the term "ECMA Script", which is the official name of the language, as specified by the *European Computer Manufacturers Association* (ECMA). As for the term *Bad smell*, we use the terms "code smell", "design smell", and "architecture smell" as synonyms. With these synonyms, we seek to expand the scope of our study to the strands of studies on *bad smells* related to software architecture, design, or source code. Thus, the resulting search string for our mapping was as follows:

("bad smell*" OR "code smell*" OR "design smell*" OR "architecture smell*") AND ("Javascript" OR "Java script" OR "ecma script" OR "ecmascript")

This search string was used in the Scopus (Elsevier) database to collect the data for our mapping study. The search engine of Scopus was configured to analyze the search string based on the title of the article, abstract content, and keywords. The data collection was performed in May-2021.

2.3 Inclusion and Exclusion Criteria

The definition of the inclusion and exclusion criteria was guided by the theme, language, number of pages, type of publication, and publication year of the items returned by the search engine. Thus, the inclusion criteria aim to select works that: (I1) focus on *bad smells*; (I2) analyze systems implemented using the Javascript programming language; (I3) were published book chapters, conference proceedings, or journals; (I4) were written in English. Moreover, The exclusion criteria aim to exclude works: (E1) With less than four pages; (E2) Published before 2000.

These criteria take into account our topic of interest (*bad smells* in JS), works published in peer-reviewed venues, in English, and after 2000 (included), since the first popular stable version of Javascript was launched only around the year 2000.

After executing the search string in the search engine, the exclusion criteria were first applied to the returned items. Then, the inclusion criteria were applied to the title and abstract of the works that were not excluded in the previous step.

Finally, for each work that passed the inclusion criteria, we applied the *backward snowballing* technique [13] to its list of references, applying the same exclusion and inclusion criteria.

3 DATA ANALYSIS

Executing the search string defined in Section 2 in the search engine of the Scopus (Elsevier) database, a total of 16 items were returned. After applying the inclusion and exclusion criteria, 8 works were considered for analysis [1, 7, 14, 16, 19, 20, 22, 24]. The other 8 items returned have been discarded; 7 of them because they are not works, but documents with the index of the annals of congresses where the works were published. A single paper violated the page exclusion criteria as it had only 3 pages. We applied the *snow balling* technique to the 8 works considered for analysis, but no new work was identified.

In the following sections, we present the analysis that answer our research questions.

3.1 List of Javascript Bad Smells

To answer the first research question of our study, we analyzed each of the selected papers identifying which *bad smells* each paper defined in the context of Javascript. It was observed if there was at least one mention in the paper that reported a *bad smell* in the Javascript language. Some data aggregation were carried out on the data extracted from the papers for the identification and counting of bad smells:

- Bad smells referring to classes or objects were aggregated together, since depending on the Javascript version, objects were used to interpret classes in the language.
- Bad smells that included code complexity were aggregated in the Cyclomatic complexity count.
- We list the bad smells "Double code" and "Unused code" as synonyms for "Duplicated code" and "Unused declaration", respectively.

Based on our analyses, a total of 26 distinct *bad smells* were detected, which are presented in Table 1. Of the 26 bad smells identified in the 8 works analyzed, the bad smells "Chained Callbacks" (6 occurrences), "Long Method" (5 occurrences), and "Switch Statement" (4 occurrences) were most frequently observed. These *bad smells* are also at the top of the incidence list of smells occurrence in other works [8, 23].

We observed that most smells identified are related to code elements with extreme size or complexity – 11 out of the 26 smells identified reside in this category: *Cyclomatic complexity*, *Depth*, *Duplicated Code*, *Extra Bind*, *Large/God/Brain Class/Object*, *Lengthy Lines*, *Long Method*, *Long Parameter List*, *Refused Bequest Spaghetti Code*, and *Switch Statement*. So there seems to be, so far, more attention to this type of problem.

We also noticed the presence of Javascript context-specific bad smells – *Extra Bind* and *This Assign*. Such smells handle errors in the scope of the language and its functionality, as in the case of *bind* and *this*.

We also observed that 4 smells are indicators of actual defects in the source code – *Argument Count Mismatch*, *Argument Type Mismatch*, *Empty Catch* and *Negative Array Index*. Therefore, these *bad smells* might be more problematic than the others.

Moreover, there are only two *bad smells* related to the way methods or functions collaborate to perform specific functionality. *Bad smells* in this category, like *Intensive Coupling*, *Disperse Coupling*, or *Shotgun Surgery* [15] are commonly investigated in other works

Table 1: List of Javascript Bad Smells

Bad Smell	Definition	Papers
Chained/Nested Callbacks	When a function call implies multiple other external function calls nested within its scope	[1] [7] [14] [16] [19] [24]
Long Method	A class method has big and/or complex definitions ahead of his main responsibility	[7] [14] [16] [19] [22]
Cyclomatic complexity	A code that has a cyclic and holds extreme complexity using the language artifacts	[1] [14] [19] [20]
Duplicated Code	A code that is often duplicated into other areas of the application	[1] [7] [20] [22]
Switch Statement	A complex Switch statement or a sequence of If statements	[7] [14] [19] [22]
Large/God/Brain Class/Object	The names brain, large, and god class/object refer to a class that is somehow connected with other classes of the project that are previously independent	[1] [7] [22]
Lengthy Lines	A line of the code that has an expressive amount of characters	[14] [16] [24]
Long Parameter List	A function or class that has an expressive amount of parameters	[14] [19] [24]
Spaghetti Code	Pejorative phrase for unstructured and difficult-to-maintain source code	[20] [22] [24]
Depth	Smell occurs when the number of nested blocks of code (or the level of indentation) is too high.	[14] [19]
Extra Bind	When you bind a function multiple times in the same code scope	[14] [19]
This Assign	When the this is reassigned to force code scope change.	[14] [19]
Undeclared Variables	Variables that are called but not yet declared in code scope (some languages like JS support use of variables that are called before defined).	[1] [7]
Unused declaration	A declared variable or function that is not used	[1] [7]
Unreachable Code	A code that is not reached because is below a break or return statement	[1] [7]
Argument Count mismatch	When a function is called with an incorrect number of arguments	[1]
Argument Type mismatch	When a function is called with the incorrect type of arguments	[1]
Array length Assignment	When a length array property is changed after definition	[1]
Closure	Long scope chaining and/or *this* operations inside closures	[7]
Empty catch	TryCatch statement with empty or missing catch fallback for error handling	[7]
Feature Envy	A class, function, or object tries to implement a feature that is already implemented in another block of code	[22]
Global variable	Use of global variables within the code, making it difficult to observe their content in the numerous system scopes and routines.	[7]
Middle Man	When a class exists as an intermediator to call a feature that is already in another class	[22]
Negative Array index	When the code tries to get a negative unit of the array	[1]
Primitive Property Assignment	Use of primitive types to assign a variable	[1]
Refused Bequest	This smell occurs when a subclass uses only a few methods of its inherited superclass, making substitution impossible.	[7]

[21], but no similar smell was identified in our mapping study. The only smells identified in our mapping study in this category are *Chained/Nested Callbacks*, which refers to cases where an excessive number of callback functions are chained or nested, and *Closure* which includes couplings within closures. Another observation we make is the lack of *bad smells* related to problems with inheritance and class abstraction, such as the classic smells *Refused Parent Bequest* and *Tradition Breaker* [15]. Although Javascript currently supports object-oriented programming, the concept of classes as we know them in other programming languages was only officially defined in JS in version 5 of Ecmascript [3], which was launched in 2015; before that, they were previously used in the form of objects and functions to adapt the language to the OO paradigm. This fact may explain why works published close to or before the year 2015 do not have information about *bad smells* related to the object-oriented programming paradigm, starting its first mention in the work of Saboury [19], in 2017.

Finally, it is worth mentioning that it was not observed framework-specific *bad smells*. Much of the popularity that the Javascript language has received in recent years is due to the growing popularity of frameworks and runtime environments (RTE) aimed at developing web systems, both those aimed at back-end development, such as NodeJS, and for front-end development, such as ReactJS, VueJS, Angular and Svelte. Despite the growing popularity of these tools, our mapping study about *bad smells* in JS did not identify any smell specifically defined for framework or RTE concepts. The definition of *bad smells* specific to frameworks and programming languages has been explored in other works, such as Kotlin and the Android ecosystem [10, 12], Python and its web framework Django [6] and Java and its web framework Spring-MVC [2]. The targeting of works aimed at *bad smells* specific to frameworks in the Javascript language is still non-existent and demonstrates a possible theme of study in the future.

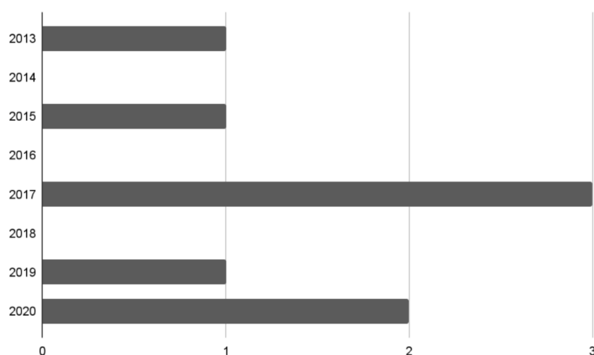

Figure 1: Works Addressing *Bad smells* in Javascript Over Time

3.2 Interest in the bad smells topic in JS over the years

To answer the second research question of our mapping study, we collected the year of publication of the papers that passed the inclusion and exclusion criteria to observe the incidence of publications on the topic of interest each year. The line chart shown in Figure 1 presents the count of works in our data set by year of publication, starting from the year the first paper was published, in 2013 [7] to the last paper found, published in 2020 [22].

There is an average of 1.125 articles per year. Considering the period between 2013 and 2020, there was no paper about *bad smells* in Javascript in the years 2014, 2016, and 2018. So there is no growth trend or increasing interest in this topic in recent years, despite the increasing popularity of Javascript and increasing interest for investigations regarding *bad smells*.

It was also observed the emergence of *bad smells* being adapted to Javascript as the year of publication progressed, such as *Middle Man* and *Feature Envy*, found only in the work of Almashfi, published in 2020 [1], and which may indicate new areas of interest in the language such as the structures of previously mentioned OO paradigm, promoting a greater incidence of smells not seen at the time of construction of this work.

4 THREATS TO VALIDITY

In this section, we discuss the threats to the validity of our study according to the guidelines presented by Petersen, Vakkalanka, and Kuzniarz for conducting systematic mappings in Software Engineering [17].

Descriptive validity. This threat relates to the extent to which observations are accurately and objectively described. In the context of our mapping study, this threat regards mainly to researcher bias, which could be present in the selection and extraction of data from the mapped studies. We mitigated this threat by having one researcher conducting the data extraction using a data collection form and a different researcher reviewing the extracted data.

Theoretical validity. This threat relates to the ability of properly capturing the concept that was the target of the investigation. Thus, in the context of our study, one dimension of this threat regards the studies about *bad smells* in Javascript that could have been missed by our search strategy. We may have missed some studies due to the terms used in the search string, which may have been too strict or may have missed an important keyword, and also because we only used one search engine (Scopus-Elsevier). We mitigated this threat by applying the *backward snowballing* technique [13], which led to no new study being added. We plan to expand our search space by adapting and executing our search string in other search engines in future work.

Generalizability validity. Threats to internal generalizability (generalization within a group) of our study refer to the extent to which the conclusions apply to the mapped studies. We mitigate this threat by following a unique and well-defined process for analyzing the data collected and aggregated from the mapped studies so that conclusions were solely based on data collected from the mapped studies. Threats to external generalizability (generalization between different groups) are related to the extent to which the conclusions apply to studies other than those mapped by our studies. We consider this a less relevant threat in the context of our study, as we intend to generalize only to studies in the context of bad smells and Javascript and not to other groups of studies in different programming languages.

Interpretive validity. In our study, this threat is mainly influenced by research bias. The first author of the study has worked for 4 years as a software developer and his practical experience may have influenced the discussions and conclusions inferred from the data. To alleviate this effect, the other researcher, who is an academic, discussed and revised the first author's analyzes and conclusions, seeking to keep the conclusions grounded on the collected data. Finally, this study will also undergo a thorough peer-review process that will help mitigate this threat.

Repeatability validity. To mitigate the threats to the repeatability of our study, we strictly followed the procedures defined by Petersen, Vakkalanka, and Kuzniarz[17] and we detail as much as possible the process we follow.

5 RELATED WORKS

The most similar related work to ours is the work conducted by Sobrinho et al. [21], who performed a systematic review of the literature on *bad smells*. The authors applied the concept of the 5W's (Which, When, What, Who, Where) to conduct their systematic review about *bad smells*, but differently from us, they did not impose any restriction of programming languages. The authors analyzed the *bad smell* types, interest over time, their goals and discoveries, the influential researchers in the field, and the distribution of related articles. In their work, "Duplicated code" was the most recurrent smell among the *bad smells* analyzed. In our mapping study, this *bad smell* tied for fourth place as the most recurrent smell among those identified for JS. This can demonstrate that, despite its large occurrence, the Javascript scenario can emphasize certain smells with a greater incidence to detriment of others more common in languages with a different focus. An example case is on smells related to classes, a fact that cannot be intrinsically observed in works focusing on multiple languages as seen in Sobrinho et al. in which only 10 of the smells found are also mentioned in the respective

work. This observation can justify the absence of the remaining bad smells because they are focused on the JS environment or have different nomenclature among the authors.

Another work related to ours is the work carried out by Fernandes et al.[8], who performed a systematic review study on *bad smell* detection tools, addressing their techniques for obtaining smells and which types were identified by each application. In their work, a total of 89 tools were identified and only 29 were available for download. In addition, the main programming languages supported by these 89 identified tools are C++, and Java: 34 tools support bad smell detection in Java and 24 tools in C++. To the detection of bad smells in Javascript, it was observed a total of 13 tools with support for this language, being the third highest occurrence in the work in question, representing 15% of the tools. Despite keeping a considerable number of tools, and, together with our study of the growing interest in bad smells in the language, it is possible to observe that the language still needs a greater growth in the tools to support the detection of smells and a greater number of works focusing on the area in question, when compared to other programming languages.

6 CONCLUSION

We conducted a mapping study about *bad smells* in the Javascript programming language. The main objective of the work was to observe which types of bad smells appear more frequently in studies related to language and analyze the growth of the theme over the years.

We identified 8 works that passed the list of inclusion and exclusion criteria, from which we extracted a total of 26 *bad smells*. We observed that most smells investigated in the context of Javascript systems are "classic" *bad smells* adapted from other works; there are still few smells defined for specific constructs of the Javascript programming language. We also observed that works about *bad smells* in Javascript are still limited when compared to other languages, like Java, and no growing trend is observable in recent years.

Finally, we intend to expand this mapping study by adapting our search string and executing it in other search engines and by answering new research questions. We also plan to conduct future works on the definition of new *bad smells* related to specific constructs of the Javascript language and investigate the negative impacts that these smells might have in real systems.

REFERENCES

[1] Nabil Almashfi and Lunjin Lu. 2020. Code Smell Detection Tool for Java Script Programs. In *5th International Conference on Computer and Communication Systems (ICCCS)*. 172–176.

[2] M. Aniche, G. Bavota, C. Treude, M. A. Gerosa, and A. van Deursen. 2018. Code smells for Model-View-Controller architectures. *Empirical Software Engineering* 23, 4 (2018), 2121–2157.

[3] Vincenzo Arceri, Isabella Mastroeni, and Sunyi Xu. 2020. Static analysis for ECMAscript string manipulation programs. *Applied Sciences (Switzerland)* 10, 9 (2020), 148.

[4] Ajay Bandi, Byron J. Williams, and Edward B. Allen. 2013. Empirical evidence of code decay: A systematic mapping study. In *20th Working Conference on Reverse Engineering (WCRE)*. 341–350.

[5] Aloisio S. Cairo, Glauco de F. Carneiro, and Miguel P. Monteiro. 2018. The impact of code smells on software bugs: A systematic literature review. Issue 11. https://doi.org/10.3390/info9110273

[6] R. Correia and E. Adachi. 2019. Detecting design violations in django-based web applications. In *10th ACM International Conference Proceeding Series (ACM)*. 33–42.

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

[8] Eduardo Fernandes, Johnatan Oliveira, Gustavo Vale, Thanis Paiva, and Eduardo Figueiredo. 2016. A review-based comparative study of bad smell detection tools. In *20th ACM International Conference Proceeding Series (ACM)*. 458.

[9] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley.

[10] A. Gong, Y. Zhong, W. Zou, Y. Shi, and C. Fang. 2020. Incorporating Android Code Smells into Java Static Code Metrics for Security Risk Prediction of Android Applications. In *20th International Conference on Software Quality, Reliability, and Security (QRS 2020)*. 30–40.

[11] Sharath Gude, Munawar Hafiz, and Allen Wirfs-Brock. 2014. JavaScript: The used parts. In *12th ACM International Conference Proceeding Series (ACM)*. 466–475.

[12] G. Hecht, N. Moha, and R. Rouvoy. 2016. An empirical study of the performance impacts of Android code smells. In *16th Proceedings - International Conference on Mobile Software Engineering and Systems, MOBILESoft 2016 (ICMESES)*. 59–69.

[13] Samireh Jalali and Claes Wohlin. 2012. Systematic literature studies: database searches vs. backward snowballing. In *Proceedings of the 2012 ACM-IEEE international symposium on empirical software engineering and measurement*. IEEE, 29–38.

[14] David Johannes, Foutse Khomh, and Giuliano Antoniol. 2019. A large-scale empirical study of code smells in JavaScript projects. *Software Quality Journal* 27, 20 (2019), 1271–1314.

[15] Michele Lanza and Radu Marinescu. 2007. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems.* Springer Science & Business Media.

[16] Frolin S. Ocariza, Karthik Pattabiraman, and Ali Mesbah. 2017. Detecting unknown inconsistencies in web applications. In *32nd IEEE/ACM International Conference on Automated Software Engineering (IEEE/ACM 2017)*. 566–577.

[17] Kai Petersen, Sairam Vakkalanka, and Ludwik Kuzniarz. 2015. Guidelines for conducting systematic mapping studies in software engineering: An update. In *3rd Information and Software Technology Symphosium (IST)*. 1–18.

[18] Václav T. Rajlich and Keith H. Bennett. 2000. A staged model for the software life cycle. *Computer* 33, 7 (2000), 66–71.

[19] Amir Saboury, Pooya Musavi, Foutse Khomh, and Giulio Antoniol. 2017. An empirical study of code smells in JavaScript projects. In *5th International Conference on Computer and Communication Systems (ICCCS 2020)*. 294–305.

[20] Ian Shoenberger, Mohamed Wiem Mkaouer, and Marouane Kessentini. 2017. On the use of smelly examples to detect code smells in JavaScript. In *20th European Conference on the Applications of Evolutionary Computation (EvoApplications 2017)*. 20–34.

[21] Elder Vicente De Paulo Sobrinho, Andrea De Lucia, and Marcelo De Almeida Maia. 2021. A Systematic Literature Review on Bad Smells-5 W's: Which, When, What, Who, Where. *IEEE Transactions on Software Engineering* 47, 32 (2021), 17–66.

[22] Amjed Tahir, Jens Dietrich, Steve Counsell, Sherlock Licorish, and Aiko Yamashita. 2020. A large scale study on how developers discuss code smells and anti-pattern in Stack Exchange sites. *Information and Software Technology* 125, 30 (2020), 256.

[23] Gustavo Vale, Eduardo Figueiredo, Ramon Abilio, and Heitor Costa. 2014. Bad smells in software product lines: A systematic review. In *8th Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)*. 84–94.

[24] Xiao Xiao, Shi Han, Charles Zhang, and Dongmei Zhang. 2015. Uncovering JavaScript Performance Code Smells Relevant to Type Mutations. , 335-355 pages. https://doi.org/10.1007/978-3-319-26529-2_18