Multi-faceted Code Smell Detection at Scale using DesigniteJava 2.0

Tushar Sharma
tushar@dal.ca
Dalhousie University, Canada

ABSTRACT

Code smell detection tools not only help practitioners and researchers detect maintainability issues but also enable repository mining and empirical research involving code smells. However, current tools for detecting code smells exhibit notable shortcomings, such as limited coverage for a diverse kind of smells at varying granularities, lack of maintenance, and inadequate support for large-scale mining studies. To address the limitations, the first major version of DesigniteJava supported code smells detection at architecture, design, and implementation smells along with commonly used code quality metrics. This paper presents DesigniteJava 2.0 that adds testability and test smell detection support. Also, the tool offers new analysis modes, including an *optimized* multi-commit analysis mode, to support large-scale multi-commit analysis. We show that the optimized multi-commit mode reduces analysis time by up to 46% without compromising the analysis efficacy. The tool is available online. Replication package including all the validation data and scripts can be found online [27]. Demonstration video can be found on YouTube.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**; **Software maintenance tools**.

KEYWORDS

Code smell detection tool, repository mining.

ACM Reference Format:
Tushar Sharma. 2023. Multi-faceted Code Smell Detection at Scale using DesigniteJava 2.0. In *Proceedings of 21st International Conference on Mining Software Repositories (MSR 2024).* ACM, New York, NY, USA, 5 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Code smells indicate the presence of quality issues, typically affecting maintainability negatively [8, 32]. High smell density in a software reduces the code quality and hampers the system's evolution. Given the importance and proliferation of code smells, researchers in the field have explored various aspects of the metaphor, including their causes, impacts, and detection methods [32].

Code smell detection tools are the heart of the majority of exploratory and empirical studies involving code smells. Researchers, as well as industrial vendors, have developed a variety of code smell detection tools [31, 32]. These tools can be divided into five categories [32]—*metric-based* [15, 38], *rule/Heuristic-based* [17, 30], *history-based* [9, 21], and *optimization-based* [20, 25] smell detection methods. Despite these efforts, the smell detection tools exhibit considerable deficiencies. First, existing code smells detection tools support detecting a limited number of smells [22, 24]. Specifically, a handful of tools (specifically, 13%—six out of 45) investigated in a study [32] detect ten or more smells. Lack of support for a wide range of code smells poses a challenge to empirical studies on smells; identifying a very small subset of smells and using the data to correlate other software engineering aspects (such as the number of bugs) makes such studies incomplete or even incorrect. Second, Code smells may arise at different granularities (*e.g.,* architecture, design, and implementation) and different artifacts (*e.g.,* production, infrastructure, and test code). Though there have been some attempts to detect, for example, architecture smells [7] and test smells [23], a comprehensive tool supporting smell detection at different granularities and artifacts is missing. Next, the majority of research prototypes are either not available online or not maintained. For example, out of six tools that support the detection of ten or more smells, only two (JSNose [6] and JSpIRIT [37]) are available online at the time of writing this text. Furthermore, both the available tools are not updated in years; JSNose and JSpIRIT were last updated ten and five years ago, respectively. Finally, for any large-scale mining study, researchers need to analyze many repositories, often all the commits. However, the available tools typically do not provide native support for carrying out such large-scale mining analysis.

To address the gap, we first introduced DesigniteJava [26, 30]. The tool supported the detection of seven architecture smells [31], 20 design smells, and 10 implementation smells [29] along with various code quality metrics. In the last seven years, we maintained the tool, fixed bugs, and added new features. We use the tool in our research [28, 31], but also made the tool available free and accessible for academic use. The community has used the tool extensively for their code smells-related research [5, 19, 36]. Our free academic license has been used by at least 175 universities[1] worldwide for education and research at the time of writing this text.

This paper presents DesigniteJava 2.0—an improved code smells detection tool that supports testability and test smells, in addition to previously supported code smells. Furthermore, the new version introduces not only native support to analyze all or a customized set of commits of a repository but also implements an optimization to improve the analysis time of large repositories. These significant improvements may further facilitate repository mining studies and empirical studies involving code smells in the field.

[1] https://www.designite-tools.com/acad-lic-request/

MSR 2024, April 2024, Lisbon, Portugal
2023. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

2 DESIGNITEJAVA 2.0

In this section, we elaborate on the tool's architecture and the newly added support for testability and test smells. Additionally, we elaborate on the optimized support to analyze multiple commits for a repository.

2.1 Tool architecture

Figure 1 shows the architecture of the tool. DesigniteJava utilizes Eclipse Java Development Toolkit (JDT) to parse the source code, prepare ASTs, and resolve symbols. The source model is the middle layer. The model invokes JDT and maintains a source code model from the information extracted from an AST with the help of JDT. The top layer of the tool contains the business logic *i.e.,* the smell detection and code quality metrics computation logic. The layer accesses the source model, identifies smells and computes metrics, and outputs the generated information in either CSV or XML files. The new version of the tool adds support to detect testability and test smells. To enable the support, we extend the existing smell detection module. We also modify the source model layer to extract additional information required for our purpose. Furthermore, the tool adds two more analyze modes—*multi-commit analysis* and *optimized multi-commit analysis*, in addition to analysis modes supported in the previous version (*analysis* and *analysis in CI* modes). These modes are specified by using command line arguments as shown in Figure 2.

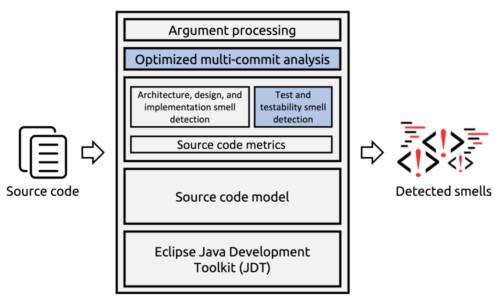

Figure 1: Architecture of DesigniteJava tool; the colored boxes represents new components for DesigniteJava2.0

Figure 2: Command line options for DesigniteJava2.0

2.2 Testability smells detection

Testability is defined as *the degree to which the development of test cases can be facilitated by the software design choices* [3, 4]. Testability smells are the programming practices that reduce the testability of a software system. The new version of DesigniteJava supports four testability smells that we summarize below.

***Hard-wired dependency*:** This smell occurs when a concrete class is instantiated and used in a class resulting in a *hard-wired dependency* [4, 11]. A *hard-wired dependency* creates tight-coupling between concrete classes and reduces the ease of writing tests for the class [4]. To detect the smell, we first detect all the objects created using the new operator in a class. Then, if the functionality of the newly created object is used (*i.e.,* at least one method is called) in the same class, we detect this smell.

***Global state*:** Global variables are, in general, widely discouraged [16]. This smell arises when a global variable or a Singleton object is used [11, 33]. Global variables introduce unpredictability and hence make tests difficult to write by developers. If a class or a field in a class is declared with `public static` modifiers, we detect this smell.

***Excessive dependency*:** This smell occurs when the class under test has excessive outgoing dependencies. Dependencies make testing harder; a large number of dependencies makes it difficult to write tests for the class under test in isolation [33, 40]. We compute *fan-out* (*i.e.,* total number of outgoing dependencies) of a class. If the fan-out of the class is more than a pre-defined threshold (customizable, by default set to 7), we detect the smell.

***Law of Demeter violation*:** This smell arises when the class under test violates the law of Demeter *i.e.,* the class is interacting with objects that are neither class members nor method parameters [34]. Violations of the law of Demeter create additional dependencies that a test has to take care of. We detect all the method invocation chains of the form `aField.get-Object().aMethod()`. We detect this smell when method calls are made on objects that are not directly associated with the current class.

2.3 Test smells detection

The tool uses the test smells definition and their detection strategies from existing studies [23, 39]. Below, we present a summary of supported test smells and the detection strategies.

Assertion roulette: We detect this smell when a test method contains more than one assertion statement without giving an explanation as a parameter in the assertion method.

Conditional test logic: We detect this smell when there is an assertion statement within a control statement block (e.g., `if` condition).

Constructor initialization: We detect this smell when a constructor of a test class initializes at least one instance variable.

Eager test: We detect this smell when a test method calls multiple production methods.

Empty test: We detect this smell when a test method does not contain any executable statement within its body.

Exception handling: We detect this smell when a test method asserts within a catch block or throws an exception, instead of using `Assert.Throws()`.
Ignored test: We detect this smell when a test method is ignored using the `Ignore` annotation.
Unknown test: We detect this smell when a test method does not contain any assert call or expected exception.

2.3.1 *Validation for testability and test smells.* We curated a *ground truth* of smells in a Java project to manually validate the tool, as explained below.
Subject system selection: We used the RepoReapers dataset [18] to select a subject system. We selected Java repositories of moderate size (between 10K and 15K), with unit-test as well as documentation ratio > 0.0, and with at least two developers. We applied the criteria and sorted the list by the number of stars. We obtained *j256/ormlite-jdbc*, *paul-hammant/paranamer*, and *forcedotcom/wsc* as the top three projects satisfying our criteria. The majority of the source code belonging to *j256/ormlite-jdbc* and *paul-hammant/paranamer* was in test cases. Hence, we selected *j256/ormlite-jdbc*, as our subject system for test smells validation. However, such repositories were unsuitable for validating testability smells since we detect testability smells in non-test code. Hence, we selected forcedotcom/wsc, a project that offers a high-performance web service stack for clients, as our subject system for the manual validation of testability smells.
Validation protocol: Two evaluators manually examined the source code of the selected subject systems and documented the testability and test smells that they found. Both evaluators hold a PhD in computer science and have over five years of software development experience. Before the evaluation, they were introduced to testability and test smells. They were allowed to use IDE features (such as "find", "find usage" (of a variable) and "find definition" (of a class) and external tools to collect code quality metrics to help them narrow their search space. Both evaluators carried out their analyses independently. It took approximately three full workdays to complete the manual analysis. After completing their manual analysis, they matched their findings to spot any differences. We used *Cohen's Kappa* [2] to measure the inter-rater agreement between the evaluators. The obtained result, 89% and 93% respectively, for testability and test smells shows a strong agreement between the evaluators. The evaluators discussed the rest of their findings and resolved the conflicts.
Validation results: We used our tool on the subject systems and identified testability and test smells. We manually matched the ground truth prepared by the evaluators and tool's results. We classified each smell instance as true positive (TP), false positive (FP), and false negative (FN). We computed precision and recall metrics using the collected data.

Table 1 presents the results of the manual evaluation for testability smells. The tool identified 161 instances of testability smells out of 172 manually verified smell instances. The tool produced two false positive instances and eleven false negative instances. The false positive instances were detected mainly because the tool identified the *hard-wired dependency* even when an object was instantiated in a method call statement. Similarly, the tool reported false negatives due to an improper resolution of enumeration types; we traced back the inconsistent behavior to the JDT parser library.

Table 1: Results of manual validation for testability smells; MVI stands for Manually Verified Instances

Testability Smells	MVI	TP	FP	FN
Hard-wired dependencies	64	63	2	1
Global state	22	22	0	0
Excessive dependencies	20	19	0	1
Law of Demeter violation	66	57	0	9
Total	**172**	**161**	**2**	**11**

Table 2: Results of manual validation for test smells; MVI stands for Manually Verified Instances

Testability Smells	MVI	TP	FP	FN
Assertion roulette	214	212	0	2
Conditional test logic	11	11	0	0
Constructor initialization	0	0	0	0
Eager test	13	13	0	0
Empty test	0	0	0	0
Exception handling	3	2	0	1
Ignored tests	2	2	0	0
Unknown test	58	58	0	0
Total	**301**	**298**	**0**	**3**

The precision and recall of the tool for testability smells based on the analysis is $161/(161+2) = \mathbf{0.99}$ and $161/(161+11) = \mathbf{0.94}$, respectively. Similarly, Table 2 shows the results of the manual evaluation carried out for test smells. Out of 301 test smells in 428 test methods, the tool correctly detected 298 smells. The cause of three instances of false negatives is traced back to the inconsistent behavior of the parser library. The precision and recall of the tool for test smells based on the analysis are $298/(298+0) = \mathbf{1.0}$ and $298/(298+3) = \mathbf{0.99}$, respectively.

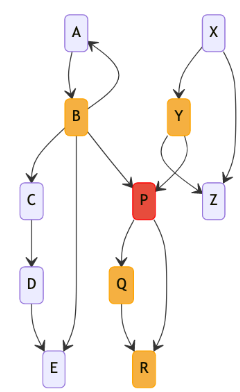

Figure 3: Dependency graph showing changed file (marked with red) and selected file for reanalysis (files marked with red and orange colors)

2.4 Optimized support for mining repositories

Often software engineering researchers need to analyze all commits of a repository, for example, to analyze code quality trends.

Researchers write programs to check out all commits individually for a repository and analyze the code using tools such as DesigniteJava. To reduce the effort, the new version of DesigniteJava introduces a new analysis mode referred to as *multi-commit* analysis to analyze multiple commits in a branch by specifying option `-ac`. For example, the command `java -jar DesigniteJava.jar -i ./myProject -o ./myProject/analysis -ac "main"` will analyze all the commits in the `main` branch of `myProject` repository. Furthermore, we may specify a range of commits to analyze using `-fr` (from-commit) and `-to` (to-commit) options. Details about various command options for the tool can also be found online[2].

However, each commit typically only changes for a small fraction of files. Despite that DesigniteJava, using the option described above *i.e.,* `-ac`, and other similar tools, analyze each commit from scratch. Hence, the tools incur significantly more computing resources as well as time by not utilizing the significant similarity in the source code from commit to commit. The new version of the tool introduces another analysis mode *viz. optimized multi-commit* analysis to overcome this limitation by utilizing the source code information from the previous commit. This mode can be invoked using the option `-aco`. The optimized mode differs from the multi-commit analysis mode in one significant way—the optimized version reuses abstract syntax tree as well as source code model and computed metrics information from the previous commit. The tool uses git utility methods to determine the changed and deleted files in a commit. The tool finds the associated source code entities for this modified file set and marks them for updation. Each class depends on other classes; the dependencies among classes must be taken into account for accurate analysis. The new version of the tool determines an *impact set* for each modified class; this impact set includes the direct dependencies, incoming and outgoing, for the class. Figure 3 presents an example. The figure shows a dependency graph among the classes of a project. If class `P` is modified, then classes `B, Y, Q, R` are considered as the impact set. The tool discards the source code information inherited from the previous commit related to the modified classes and their associated impact set and rebuilds the source code model for them. In this way, the tool attempts to optimize the code analysis without compromising the accuracy of the tool.

Evaluation: We evaluate the accuracy of the new optimized analysis mode by comparing the produced output with and without optimization. We use a script that we developed (available in the replication package) to compare the outcome of both the analysis modes. We also measure analysis time in both cases. For this evaluation, we search Java repositories within the Apache organization with minimum 10,000 commits, minimum 100 issues, minimum 200 contributors, and a minimum 10,000 stars. The criteria gave us three open-source repositories—Druid[3], Pulsar[4], and ShardingSphere[5]. We analyze first 1,000 commits in both the modes. Table 3 presents the evaluation results. Comparing the results for both the modes for the selected repositories show identical results. However, we observe that the **optimized multi-commit analysis mode achieves the same results by consuming up to 46% less time than the multi-commit analysis mode.** With this optimization the tool saves significant research time and computing resources.

[2]https://www.designite-tools.com/docs/index.html

[3]https://github.com/apache/druid

[4]https://github.com/apache/pulsar

[5]https://github.com/apache/shardingsphere

Table 3: Analysis time (in seconds) for the selected commits using *ac* (multi-commit) mode compared to *aco* (optimized multi-commit) mode

Repository	Analysis time (*ac* mode)	Analysis time (*aco* mode)	Efficiency gain
Druid	14,722	7,922	**46.2%**
Pulsar	52,789	28,124	**46.7%**
ShardingSphere	10,580	6,300	**40.5%**

3 RELATED WORK

Software engineering research contains a large body of work related to code smell detection. Smell detection approaches can be divided into five categories [32]. *Metric-based smell detection methods* [15, 38] compute a set of source code metrics and detect smells by applying appropriate thresholds [15]. *Rule/Heuristic-based smell detection* methods [17, 35] define rules/heuristics to detect code smells. *History-based smell detection* approaches observe the evolutionary properties in source code [9, 21] to infer smells in the code. *Optimization-based smell detection* approaches [20, 25] use optimization algorithms typically on code quality metrics to detect smells in a given source code. In recent times, machine-learning (ML) techniques have been applied extensively to detect code smells. The *ML-based smell detection* methods, typically identify a set of features (such as code quality metrics) and use them to train a model. Early approaches used traditional ML, such as Bayesian and support vector machine, and a fixed set of code quality metrics as features to classify smelly code snippets from benign ones [1, 12, 14]. Several studies use deep learning techniques to identify code smells [10, 13, 28].

There have been a few tools to detect test smells. JNose [39] detects 21 test smells and analyzes the quality evolution of a software project. Similarly, TsDetect [23] supports detecting 19 test smells. However, first, existing test smell detection tools are not integrated with traditional code smells increasing the number of tools required for code quality analysis for a software development team. Also, the existing tools are not suitable for large-scale empirical analysis. For example, analyzing code using TsDetect involves a manual step requiring mapping test files and corresponding production files. The proposed tool addresses both limitations and provides a comprehensive code smell detection tool for Java.

4 CONCLUSIONS

DesigniteJava has served the software engineering community for the last seven years. The tool supports the detection of a large number of architecture smells, design smells, and implementation smells, along with various code quality metrics. We presented a new, improved version of the tool that adds support for testability and test smells detection. Another significant addition to the tool is supporting optimized multi-commit analysis of a repository. These significant improvements will provide additional automated tool support to software developers and further facilitate repository mining and empirical studies related to code smells.

REFERENCES

[1] Francesca Arcelli Fontana, Mika V. Mäntylä, Marco Zanoni, and Alessandro Marino. 2016. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering* 21, 3 (01 Jun 2016), 1143–1191. https://doi.org/10.1007/s10664-015-9378-4

[2] Kenneth J. Berry and Jr. Paul W. Mielke. 1988. A Generalization of Cohen's Kappa Agreement Measure to Interval Measurement and Multiple Raters. *Educational and Psychological Measurement* 48, 4 (1988), 921–933. https://doi.org/10.1177/0013164488484007

[3] Robert V. Binder. 1994. Design for Testability in Object-Oriented Systems. *Commun. ACM* 37, 9 (sep 1994), 87–101. https://doi.org/10.1145/182987.184077

[4] Vishal Chowdhary. 2009. Practicing Testability in the Real World. In *Intl. Conference on Software Testing Verification and Validation*. 260–268. https://doi.org/10.1109/ICST.2009.53

[5] André Eposhi, Willian Oizumi, Alessandro Garcia, Leonardo Sousa, Roberto Oliveira, and Anderson Oliveira. 2019. Removal of Design Problems through Refactorings: Are We Looking at the Right Symptoms?. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. 148–153. https://doi.org/10.1109/ICPC.2019.00032

[6] Amin Milani Fard and Ali Mesbah. 2013. JSNOSE: Detecting javascript code smells. In *IEEE 13th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 116–125.

[7] Francesca Arcelli Fontana, Ilaria Pigazzini, Riccardo Roveda, Damian Tamburri, Marco Zanoni, and Elisabetta Di Nitto. 2017. Arcan: A Tool for Architectural Smells Detection. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. IEEE, 282–285.

[8] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Programs* (1 ed.).

[9] Shizhe Fu and Beijun Shen. 2015. Code Bad Smell Detection through Evolutionary Data Mining. In *International Symposium on Empirical Software Engineering and Measurement*. IEEE, 41–49.

[10] Mouna Hadj-Kacem and Nadia Bouassida. 2018. A Hybrid Approach To Detect Code Smells using Deep Learning.. In *ENASE*. 137–146.

[11] Miško Hevery. 2008. Writing Testable Code. https://testing.googleblog.com/2008/08/by-miko-hevery-so-you-decided-to.html.

[12] Foutse Khomh, Stéphane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. 2009. A Bayesian Approach for the Detection of Code and Design Smells. In *QSIC '09: Proceedings of the 2009 Ninth International Conference on Quality Software*. 305–314.

[13] Hui Liu, Jiahao Jin, Zhifeng Xu, Yifan Bu, Yanzhen Zou, and Lu Zhang. 2019. Deep learning based code smell detection. *IEEE Transactions on Software Engineering* (2019).

[14] Abdou Maiga, Nasir Ali, Neelesh Bhattacharya, Aminata Sabané, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Esma Aïmeur. 2012. Support vector machines for anti-pattern detection. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 278–281.

[15] R Marinescu. 2005. Measurement and quality in object-oriented design. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*. IEEE, 701–704.

[16] LF Marshall and Jim Webber. 2000. Gotos considered harmful and other programmers' taboos. *Department of Computing Science Technical Report Series* (2000).

[17] Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Françoise Le Meur. 2010. DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Trans. Software Eng.* 36, 1 (2010), 20–36. https://doi.org/10.1109/TSE.2009.50

[18] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating GitHub for engineered software projects. *Empirical Software Engineering* 22, 6 (Dec. 2017), 3219–3253. https://doi.org/10.1007/s10664-017-9512-6

[19] Willian Oizumi, Leonardo Sousa, Anderson Oliveira, Luiz Carvalho, Alessandro Garcia, Thelma Colanzi, and Roberto Oliveira. 2019. On the density and diversity of degradation symptoms in refactored classes: A multi-case study. In *IEEE 30th International Symposium on Software Reliability Engineering*. 346–357.

[20] Ali Ouni, Raula Gaikovina Kula, Marouane Kessentini, and Katsuro Inoue. 2015. Web Service Antipatterns Detection Using Genetic Programming. In *GECCO '15: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*. ACM, 1351–1358.

[21] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. 2015. Mining version histories for detecting code smells. *IEEE Transactions on Software Engineering* 41, 5 (2015), 462–489.

[22] Fabio Palomba, Andrea De Lucia, Gabriele Bavota, and Rocco Oliveto. 2014. Anti-Pattern Detection. In *Anti-pattern detection: Methods, challenges, and open issues*. Elsevier, 201–238.

[23] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. 2020. TsDetect: An Open Source Test Smells Detection Tool. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. 1650–1654. https://doi.org/10.1145/3368089.3417921

[24] Ghulam Rasool and Zeeshan Arshad. 2015. A review of code smell mining techniques. *Journal of Software: Evolution and Process* 27, 11 (Nov. 2015), 867–895.

[25] Dilan Sahin, Marouane Kessentini, Slim Bechikh, and Kalyanmoy Deb. 2014. Code-Smell Detection as a Bilevel Problem. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 1 (Oct. 2014), 6–44.

[26] Tushar Sharma. 2018. DesigniteJava. https://doi.org/10.5281/zenodo.2566861

[27] Tushar Sharma. 2023. *DesigniteJava 2.0*. https://doi.org/10.5281/zenodo.10300873

[28] Tushar Sharma, Vasiliki Efstathiou, Panos Louridas, and Diomidis Spinellis. 2021. Code smell detection by deep direct-learning and transfer-learning. *Journal of Systems and Software* 176 (2021), 110936. https://doi.org/10.1016/j.jss.2021.110936

[29] T. Sharma, M. Fragkoulis, and D. Spinellis. 2017. House of Cards: Code Smells in Open-Source C# Repositories. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 424–429. https://doi.org/10.1109/ESEM.2017.57

[30] Tushar Sharma, Pratibha Mishra, and Rohit Tiwari. 2016. Designite — A Software Design Quality Assessment Tool. In *Proceedings of the First International Workshop on Bringing Architecture Design Thinking into Developers' Daily Activities (BRIDGE '16)*. ACM. https://doi.org/10.1145/2896935.2896938

[31] Tushar Sharma, Paramvir Singh, and Diomidis Spinellis. 2020. An empirical investigation on the relationship between design and architecture smells. *Empirical Software Engineering* 25, 5 (2020), 4020–4068.

[32] Tushar Sharma and Diomidis Spinellis. 2018. A survey on software smells. *Journal of Systems and Software* 138 (2018), 158 – 173. https://doi.org/10.1016/j.jss.2017.12.034

[33] Girish Suryanarayana, Ganesh Samarthyam, and Tushar Sharma. 2014. *Refactoring for Software Design Smells: Managing Technical Debt* (1 ed.). Morgan Kaufmann.

[34] David Thomas and Andrew Hunt. 2019. *The Pragmatic Programmer: your journey to mastery*.

[35] Nikolaos Tsantalis and Alexander Chatzigeorgiou. 2011. Identification of Extract Method Refactoring Opportunities for the Decomposition of Methods. *Journal of Systems & Software* 84, 10 (Oct. 2011), 1757–1782. https://doi.org/10.1016/j.jss.2011.05.016

[36] Anderson Uchôa, Caio Barbosa, Willian Oizumi, Publio Blenilio, Rafael Lima, Alessandro Garcia, and Carla Bezerra. 2020. How Does Modern Code Review Impact Software Design Degradation? An In-depth Empirical Study. In *IEEE International Conference on Software Maintenance and Evolution*. 511–522. https://doi.org/10.1109/ICSME46990.2020.00055

[37] Santiago Vidal, Hernan Vazquez, J Andrés Díaz-Pace, Claudia Marcos, Alessandro Garcia, and Willian Oizumi. 2016. JSpIRIT: A flexible tool for the analysis of code smells. In *Proceedings - International Conference of the Chilean Computer Science Society, SCCC*. IEEE, 1–6.

[38] Santiago A Vidal, Claudia Marcos, and J Andrés Díaz-Pace. 2014. An approach to prioritize code smells for refactoring. *Automated Software Engineering* 23, 3 (2014), 501–532.

[39] Tássio Virgínio, Luana Martins, Larissa Rocha, Railana Santana, Adriana Cruz, Heitor Costa, and Ivan Machado. 2020. JNose: Java Test Smell Detector. In *Proceedings of the 34th Brazilian Symposium on Software Engineering* (Natal, Brazil). 564–569. https://doi.org/10.1145/3422392.3422499

[40] YuMing Zhou, Hareton Leung, QinBao Song, JianJun Zhao, HongMin Lu, Lin Chen, and BaoWen Xu. 2012. An in-depth investigation into the relationships between structural metrics and unit testability in object-oriented systems. *Science china information sciences* 55, 12 (2012), 2800–2815.