TD Classifier: Automatic Identification of Java Classes with High Technical Debt

Dimitrios Tsoukalas
Centre for Research and Technology Hellas
Thessaloniki, Greece
Department of Applied Informatics, University of Macedonia
Thessaloniki, Greece
tsoukj@iti.gr

Alexander Chatzigeorgiou
Department of Applied Informatics, University of Macedonia
Thessaloniki, Greece
achat@uom.edu.gr

Apostolos Ampatzoglou
Department of Applied Informatics, University of Macedonia
Thessaloniki, Greece
a.ampatzoglou@uom.edu.gr

Nikolaos Mittas
Department of Chemistry, International Hellenic University
Thessaloniki, Greece
nmittas@chem.ihu.gr

Dionysios Kehagias
Centre for Research and Technology Hellas
Thessaloniki, Greece
diok@iti.gr

ABSTRACT

To date, the identification and quantification of Technical Debt (TD) rely heavily on a few sophisticated tools that check for violations of certain predefined rules, usually through static analysis. Different tools result in divergent TD estimates calling into question the reliability of findings derived by a single tool. To alleviate this issue, we present a tool that employs machine learning on a dataset built upon the convergence of three widely-adopted TD Assessment tools to automatically assess the class-level TD for any arbitrary Java project. The proposed tool is able to classify software classes as high-TD or not, by synthesizing source code and repository activity information retrieved by employing four popular open source analyzers. The classification results are combined with proper visualization techniques, to enable the identification of classes that are more likely to be problematic. To demonstrate the proposed tool and evaluate its usefulness, a case study is conducted based on a real-world open-source software project. The proposed tool is expected to facilitate TD management activities and enable further experimentation through its use in an academic or industrial setting.

Video: https://youtu.be/umgXU8u7lIA
Running Instance: http://160.40.52.130:3000/tdclassifier
Source Code: https://gitlab.seis.iti.gr/root/td-classifier.git

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; *Software creation and management*; • **Computing methodologies** → *Machine learning*.

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

KEYWORDS

technical debt, technical debt identification, machine learning, tool

ACM Reference Format:
Dimitrios Tsoukalas, Alexander Chatzigeorgiou, Apostolos Ampatzoglou, Nikolaos Mittas, and Dionysios Kehagias. 2018. TD Classifier: Automatic Identification of Java Classes with High Technical Debt. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 5 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Technical Debt (TD) [7] is a metaphor commonly used to indicate quality compromises that can yield short-term benefits in the software development process, but may negatively affect the long-term quality of software. In software affected by the presence of TD, wasted effort due to TD can reach up to 23% of total developers' time [5]. However, software companies cannot afford to repay all the TD that is generated continuously [9], and therefore, effective TD management calls for appropriate tooling. The TD identification techniques adopted by most of the existing tools rely on predefined rules that can be asserted by static source code analysis techniques [3]. However, the fact that each tool uses its own rulesets to identify TD issues leads to important shortcomings affecting both academia and practice [1]. Regarding academia, the lack of a tool acting as ground truth leads to construct validity threats in empirical studies. On the other hand, practitioners are always skeptical about which tool to trust for efficient TD identification.

Given the aforementioned challenges, in our recent research work [12] we empirically evaluated statistical and Machine Learning (ML) algorithms for their ability to classify software classes as High/Not-High TD. As ground truth for the development of the proposed classification framework, we considered a "*commonly agreed TD knowledge base*" [1], i.e., an empirical benchmark of classes that exhibit high levels of TD, based on the convergence of three widely-adopted TD assessment tools, namely SonarQube [6], CAST [8], and Squore [4]. As model features we considered a wide range of software factors spanning from code metrics to repository activity, retrieved by employing four popular open source tools, namely

PyDriller [11], CK [2], PMD's Copy/Paste Detector[1] (CPD), and cloc[2]. The findings revealed that a subset of superior classifiers are able to identify TD issues with sufficient accuracy and reasonable effort, achieving an F2-measure score of approximately 0.79 with an associated Class Inspection ratio of approximately 0.10.

Based on our previous research work [12], and to demonstrate the usefulness of the proposed classification framework in practice, in this paper we introduce *TD Classifier*, a novel tool that employs Machine Learning (ML) for classifying software classes as High/Not-High TD for any arbitrary Java project, just by pointing to its git repository. The tool subsumes the collective knowledge that would be extracted by combining the results of the three aforementioned TD assessment tools and relies on four open-source tools to automatically retrieve all independent variables and yield the identified high-TD classes. In that way, it enables easy identification and further experimentation of TD issues, without having to resort to a multitude of commercial and open source tools. TD Classifier is implemented as a web application, including both a backend and its associated frontend. It offers interactive visualizations that enable the prompt identification of classes that are more likely to be problematic. In order to demonstrate our approach, we conducted a case study on an open-source software application, namely Apache Commons IO.

2 SYSTEM OVERVIEW

2.1 Methodology

This section briefly presents the "heart" of the TD Classifier tool, i.e., the methodology that was followed in our previous research study [12] in order to build the classification model that is responsible for identifying high-TD software classes. Apart from the research study per se, supporting material containing the datasets and scripts used for data collection, data preparation and classification model construction can be found online[3]. Similarly to any ML task, the followed approach consists of the familiar steps of data collection, data preparation, and model building.

Starting with the data collection step, the dataset that was used to train the TD classifier is primarily based on an empirical benchmark that was constructed in a study by Amanatidis et al. [1]. In that study, the authors examined the TD assessment capability of three leading tools (i.e., SonarQube, CAST, and Squore) on 25 Java open source projects, intending to evaluate the degree of agreement (or diversity) among them and identify profiles of classes/files sharing similar levels of TD (e.g., that of high TD levels in all employed tools). By exploiting this empirical benchmark, we labeled the software classes belonging to the high-TD level profile as "high-TD", whereas the rest of the classes were labeled as "not high-TD", establishing in that way the "ground truth" for our binary classification task. Throughout this process, we ended up with a dataset containing 18.857 classes, out of which 1.283 are labeled as high-TD.

Based on the notion that multiple sources of information will result in a more accurate model, we extended the initial dataset by building a set of 18 independent variables of different nature. Specifically, various code-related metrics (such as structural properties, size, etc.) and metrics that capture aspects of the development process (such as code churn, commits and contributors count, etc.) were considered for their effect on discriminating between high- and not-high-TD class instances. To collect these class-level metrics, a set of well-known open source tools was employed. At first, development process metrics were computed by employing PyDriller, a Python framework meant for mining Git repositories. PyDriller was used to compute class-level Git-related metrics, such as commits count, code churn, and contributors' experience across the whole evolution of each class. Moreover, three additional tools, namely CK, PMD's Copy/Paste Detector (CPD), and cloc, were used for computing code-related metrics. More specifically, CK, a tool that calculates class-level metrics in Java projects through static analysis was used to compute various OO metrics, such as CBO, DIT, and LCOM for each class. Subsequently, CPD, a tool able to locate duplicate code in various programming languages, including Java, was employed to compute the density of duplicated lines for each class. Finally, cloc, an open-source tool able to count comment lines and source code lines in many programming languages, was used to compute the total number of code and comment lines' density for each class.

After extracting the various code and development process metrics for each of the 18.857 Java classes that comprise our dataset, we proceeded with appropriate data preparation tasks, which include missing values handling, outlier detection, and oversampling techniques, to account for the class imbalance problem that was present in our dataset. In addition, within the context of feature selection, we performed a statistical exploratory analysis concluding that all metrics can discriminate and potentially be used as predictors of high-TD software classes.

The final step of the methodology included model selection, training, and performance evaluation. For this purpose, we explored a set of well-established statistical and ML algorithms that have been extensively applied in other similar experimental studies. More specifically, seven different classifiers were evaluated, including Logistic Regression, Naıve Bayes, Support Vector Machines, and Random Forest, among others. By applying a repeated stratified cross-validation process accompanied with the Scott-Knott [10] hypothesis testing, the findings of our experiments revealed that a subset of four superior classifiers can effectively identify high-TD software classes, with Random Forest being the best-performing model among them. More specifically, Random Forest achieved an *F2-measure* score of approximately 0.79, with a *recall* close to 0.85. As will be shown in Section 2.2, this pre-trained Random Forest classifier constitutes the core of the proposed tool. Its relatively high performance is expected to enable practitioners to identify candidate TD items in their own systems with a high degree of certainty that these items are indeed problematic.

2.2 Implementation

As a proof of concept, the proposed approach described in Section 2.1 has been implemented in the form of a web tool. A running instance of the tool is available online[4], enabling in that way its

[1]https://pmd.github.io/latest/pmd_userdocs_cpd.html
[2]https://github.com/AlDanial/cloc#quick-start-
[3]https://sites.google.com/view/ml-td-identification/home
[4]http://160.40.52.130:3000/tdclassifier

adoption by developers in practice, and, in turn, its further quantitative and qualitative evaluation by the community.

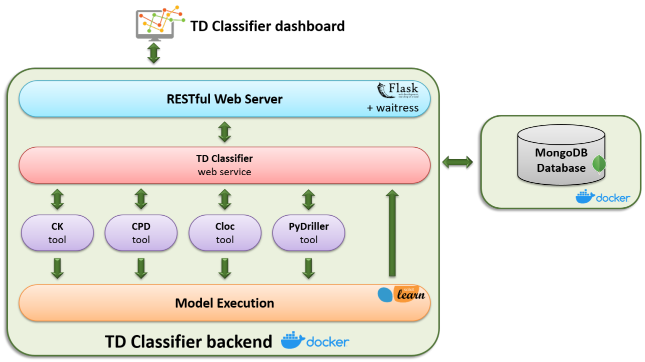

Figure 1: Overall Architecture

Figure 1 depicts the overall architecture of the TD Classifier tool. The tool is implemented in the form of a web application, including both a backend and its associated frontend. The backend of the tool, developed in Python, is actually a Microservice, making it easily accessible to the software engineering community and facilitating its integration into third-party software.

As can be seen by Figure 1, the entry point of the TD Classifier backend is a RESTful web server that uses the Flask[5] web framework wrapped inside Waitress[6], a Python WSGI production-ready server. At a lower level, the server exposes the TD Classifier API, implemented as an individual web service. This web service plays the role of an orchestrator that is responsible for: i) cloning a project, ii) invoking the analysis tools described in Section 2.1 (i.e., PyDriller, CPD, etc.) for the collection of the required metrics, iii) executing the pre-trained classifier and finally returning the results.

To facilitate the building and deployment process, Docker technology has been considered. More specifically, the tool's backend has been implemented as an individual Docker Image and deployed as an individual Docker Container. For this purpose, a Docker File, i.e., a "recipe" that describes what tools should be bundled inside the container, has been created and is available online in the repository of the tool. In that way, potential users can generate their own TD Classifier backend container easily, by building the Image from scratch and hosting it locally. In addition, apart from the scripts of the TD Classifier backend per se, all of the third-party analysis tools that are responsible for gathering the required model input are also bundled into the Docker Image as standalone executables (in the form of either jar files or shell scripts natively provided by the developers of the tool). This setup not only enhances portability by making the tool easy to install but also speeds up execution time as no external calls are required for their execution. It is worth mentioning that the analysis tools run in parallel, in order to reduce the tool's overall execution time.

Finally, a MongoDB database dedicated to storing the output of the TD Classifier web service allows the tool to quickly retrieve past results upon demand, without having to go through the time-consuming process of re-executing the analysis tools and the dedicated classifier. The database is optional and is also "dockerized" within its own container.

Apart from the tool's backend, an intuitive frontend (i.e., user interface) has been also implemented in order to facilitate its adoption in practice. The TD Classifier frontend has been integrated into the SDK4ED platform, which is the main outcome of the successful culmination of the SDK4ED[7] European project. The frontend of the tool, developed using the React[8] framework, communicates seamlessly with the backend, allowing the easy invocation of the main functionalities (i.e., web services) that the tool provides, and the visualization of the produced results. Additional information regarding the TD Classifier frontend is presented in Section 3, where we provide a case study on a real-world open-source software application that evaluates the usefulness of the proposed tool in practice.

3 EVALUATION

In this section, the proposed tool is demonstrated through a case study on a real-world open source software application. This case study also acts as a preliminary testbed for evaluating the ability of the proposed approach to identify candidate high-TD items. To evaluate the effectiveness of the TD Classifier tool, we use a popular open source Java project, namely Apache Commons IO[9]. Apache Commons IO is a library of utilities to assist with developing IO functionality, whose code is hosted on GitHub[10] with more than 3.000 commits. It should be mentioned that this project has not been used in our research study [12] for model training or evaluation.

Since TD Classifier is part of the overall SDK4ED Dashboard, the user must initially navigate to the SDK4ED Dashboard home page[11] and select an existing project, or create a new one. Then, they can navigate to the "TD Classifier" panel (located under the "Technical Debt" drop-down button on the top navigation menu), where they can select the type of analysis they would like to execute and click on the "Run Analysis" button to start the process. Currently, the tool supports three types of analyses: A *Fast* analysis will take into account only the software classes that were modified during the last 100 commits, a *Normal* analysis the classes that were modified during the last 1000 commits, whereas a *Full* analysis will take into account the whole project history.

For the sake of demonstrating the TD Classifier tool on the Apache Commons IO project, a Full analysis is selected. Once the process finishes, the user is presented with a screen that visualizes the results, as depicted in Figure 2. On the upper part of the panel, a notification informs the user that the tool has identified 26 potentially high-TD classes, out of the total 362 analyzed classes.

To effectively convey the output of the TD Classifier to the developers and project managers of the software application, a heat map has been selected as a means of visualization. As can be seen by inspecting Figure 2, the middle panel contains a heat map that presents the classification results retrieved from the analysis of the

[5] https://flask.palletsprojects.com/en/2.0.x/
[6] https://docs.pylonsproject.org/projects/waitress/en/latest/
[7] https://sdk4ed.eu/
[8] https://mdbootstrap.com/docs/react/
[9] https://commons.apache.org/proper/commons-io/
[10] https://github.com/apache/commons-io
[11] http://160.40.52.130:3000/

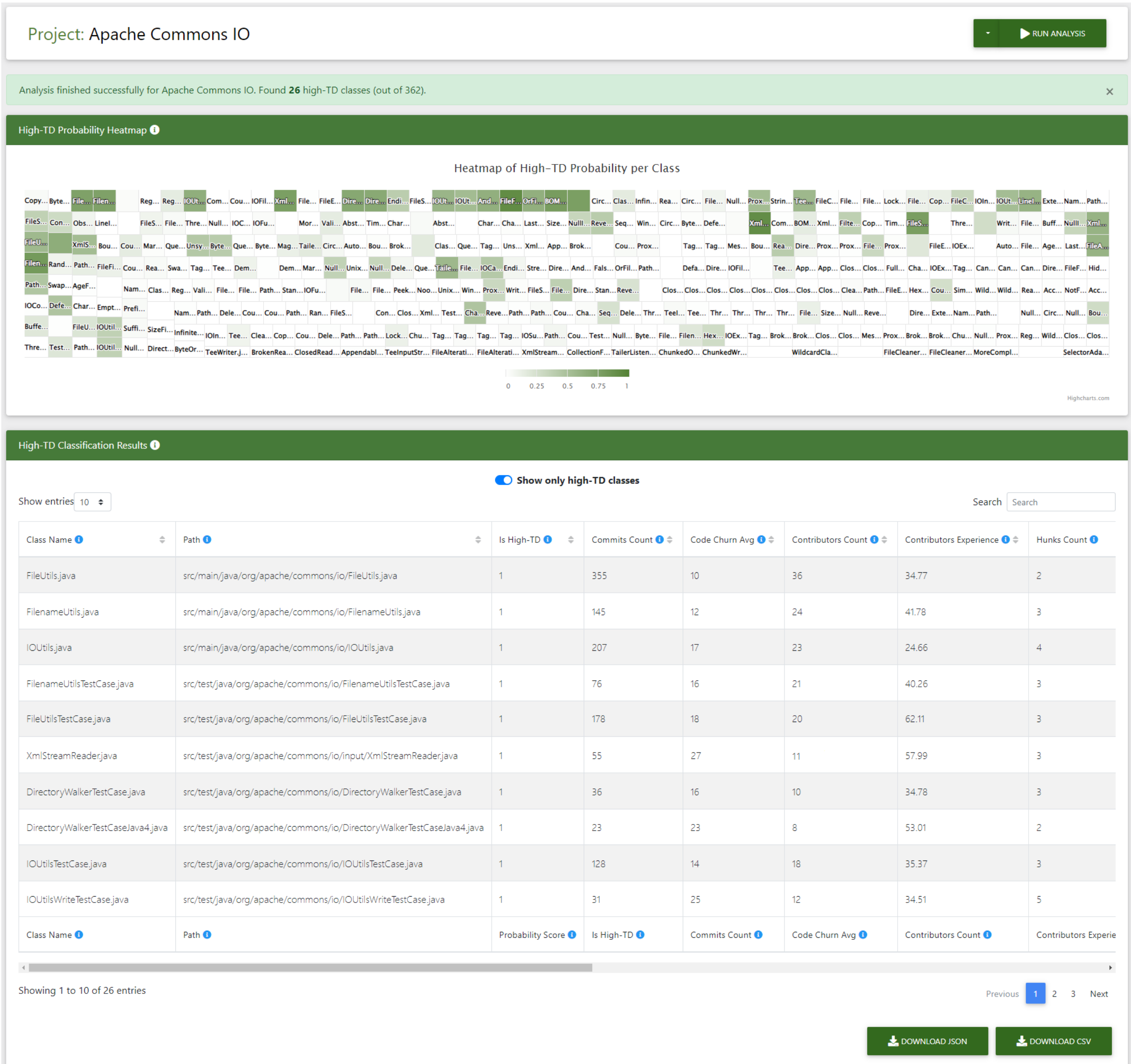

Figure 2: Heat map and complementary table visualizing the TD Classifier results for the Apache Commons IO project

Apache Commons IO project. In particular, the rectangles correspond to the classes of the selected software project, as identified and analyzed during data collection. The color of each rectangle denotes the probability of the corresponding class to be problematic (i.e., have high-TD), as calculated by the dedicated pre-trained classifier. More specifically, the greener the rectangle, the higher the probability that a class is problematic. In that way, the tool enables practitioners to promptly identify candidate TD items and therefore, plan more targeted refactoring activities.

Apart from the heat map, a complementary table comprising the detailed results of the analysis is presented at the bottom panel of Figure 2. This table contains supplementary information that, in addition to the information of whether a class is of high TD or not, includes also all of the 18 development process metrics (e.g., commits count, code churn, and contributors experience) and code metrics (e.g., CBO, DIT, and LCOM) that were calculated during the data collection process of the analysis. A toggle button at the top of the table allows the user to focus only on the classes that were identified as problematic. Moreover, through the table's sorting functionality, the user can rank the results based on any characteristic of interest, while a search field allows the easy retrieval of information for any specific class. Finally, two dedicated buttons at the bottom of the table allow the user to download the analysis results in JSON or CSV format, for further processing.

To perform a preliminary comparative analysis between the TD Classifier tool and other well-established TD assessment tools, we analyzed the Apache Commons IO project using SonarQube. SonarQube is one of the three tools that helped build the ground truth [1] that was used for the construction of our model [12] and the only one among the three tools that does not require a commercial licence. It should also be noted that SonarQube metrics are not part of our classifier's features.

As a first example, let us consider *FileUtils.java*, i.e., the first class in our list of identified high-TD classes, as presented in Figure 2. To have an indication of whether this class was correctly labeled as high-TD in the first place, we took a closer look at the analysis results produced by SonarQube. More specifically, SonarQube has ranked this class 3rd in terms of issues (22 identified), 2nd in terms of cyclomatic complexity (value of 265), and 5th in terms of TD accumulation (3 hours). In another example, let us consider the second entry in our high-TD classes list, i.e., *FilenameUtils.java*. By inspecting SonarQube results, we observed that the tool has ranked this class 6th in terms of issues (14 identified), 3rd in terms of cyclomatic complexity (value of 226), and 2nd in terms of TD accumulation (4.5 hours). Finally, let us consider the third entry in our high-TD classes list, i.e., *IOUtils.java*. By revisiting SonarQube results, we observed that it has ranked this class 2nd in terms of issues (29 identified), 1st in terms of cyclomatic complexity (value of 283), and 1st in terms of TD accumulation (5 hours). Similar observations can be also made for the rest of the classes identified as high-TD by our tool. The above comparison results provide us with preliminary evidence that the classes identified as high-TD by the TD Classifier are indeed problematic.

On the other hand, we identified cases of classes that were labeled as problematic by our tool, but at the same time their TD-related importance was probably underestimated by SonarQube. As an example, let us consider *XmlStreamReader.java*. As can be seen by inspecting the list of identified high-TD classes in Figure 2, the relatively high complexity (wmc=131), low cohesion (lcom=147), high code churn average (27), or high number of contributors (11) make this class a good high-TD candidate. On the other hand, SonarQube has labeled this class as having no TD (0 minutes), probably due to the fact that it only considers code smell issues to calculate TD remediation effort. While large-scale analysis is required to further evaluate the validity and generalizability of the above findings, our preliminary comparative analysis combined with the relatively high performance obtained through our related research work [12] highlights the practical importance of TD Classifier. Ultimately, the derived tool subsumes the collective knowledge that would be extracted by combining the results of various well-established TD tools, therefore increasing the chances that the identified classes suffer indeed from high-TD.

4 CONCLUSION

This paper introduces TD Classifier, a TD identification tool that builds upon the collective knowledge acquired by three leading TD tools and relies on open-source tools to automatically identify high-TD classes for any arbitrary Java project by pointing to its git repository. We demonstrate the tool's usefulness by a case study using the Apache Commons IO project. Our evaluation shows that TD Classifier is expected to facilitate TD management activities and enable further future experimentation through its use in an academic or industrial setting.

TD Classifier will continue to evolve to meet the challenges posed by its use in both academia and practice. We plan to evaluate the tool and report additional qualitative analysis through a large-scale case study in an industrial setting. We also plan to improve the tool's performance and scalability, as well as to extend it in other programming languages (e.g., C/C++, python, JavaScript, etc.), by incorporating additional analysis tools into the analysis pipeline.

ACKNOWLEDGMENTS

This work is partially funded by the European Union's Horizon 2020 Research and Innovation Programme through SmartCLIDE project under Grant Agreement No. 871177.

REFERENCES

[1] Theodoros Amanatidis, Nikolaos Mittas, Athanasia Moschou, Alexander Chatzigeorgiou, Apostolos Ampatzoglou, and Lefteris Angelis. 2020. Evaluating the agreement among technical debt measurement tools: building an empirical benchmark of technical debt liabilities. *Empirical Software Engineering* 25, 5 (2020), 4161–4204. https://doi.org/10.1007/s10664-020-09869-w

[2] Maurício Aniche. 2015. *Java code metrics calculator (CK).* Available in https://github.com/mauricioaniche/ck/.

[3] Paris Avgeriou, Davide Taibi, Apostolos Ampatzoglou, Francesca Arcelli Fontana, Terese Besker, Alexander Chatzigeorgiou, Valentina Lenarduzzi, Antonio Martini, Athanasia Moschou, Ilaria Pigazzini, Nyyti Saarimäki, Darius Sas, Saulo Toledo, and Angeliki Tsintzira. 2021. An Overview and Comparison of Technical Debt Measurement Tools. *IEEE Software, accepted for publication* (2021).

[4] Boris Baldassari. 2013. SQuORE: a new approach to software project assessment.. In *International Conference on Software & Systems Engineering and their Applications*, Vol. 6.

[5] Terese Besker, Antonio Martini, and Jan Bosch. 2019. Software developer productivity loss due to technical debt—A replication and extension study examining developers' development work. *Journal of Systems and Software* 156 (2019), 41–61. https://doi.org/10.1016/j.jss.2019.06.004

[6] G Ann Campbell and Patroklos P Papapetrou. 2013. *SonarQube in action* (1st edn ed.). Manning Publications Co.

[7] Ward Cunningham. 1993. The WyCash portfolio management system. *ACM SIGPLAN OOPS Messenger* 4, 2 (1993), 29–30.

[8] Bill Curtis, Jay Sappidi, and Alexandra Szynkarski. 2012. Estimating the principal of an application's technical debt. *IEEE software* 29, 6 (2012), 34–42. https://doi.org/10.1109/MS.2012.156

[9] Antonio Martini, Jan Bosch, and Michel Chaudron. 2015. Investigating Architectural Technical Debt accumulation and refactoring over time: A multiple-case study. *Information and Software Technology* 67 (2015), 237–253.

[10] A. J. Scott and M. Knott. 1974. A Cluster Analysis Method for Grouping Means in the Analysis of Variance. *Biometrics* 30, 3 (1974), 507–512. https://doi.org/10.2307/2529204

[11] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. 2018. PyDriller: Python Framework for Mining Software Repositories. In *The 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE).* https://doi.org/10.1145/3236024.3264598

[12] D. Tsoukalas, N. Mittas, A. Chatzigeorgiou, D. D. Kehagias, A. Ampatzoglou, T. Amanatidis, and L. Angelis. 2021. Machine Learning for Technical Debt Identification. *IEEE Transactions on Software Engineering* 01 (2021), 1–1. https://doi.org/10.1109/TSE.2021.3129355