IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.Doi Number

Experience with Managing Technical Debt in Scientific Software Development using the EXA2PRO framework

Nikolaos Nikolaidis[1], Dimitrios Zisis[1], Apostolos Ampatzoglou[1], Alexander Chatzigeorgiou[1] and Dimitrios Soudris[2]

[1]Department of Applied Informatics, University of Macedonia, Thessaloniki 54636, Greece
[2] School of Electrical and Computer Engineering, National Technical University of Athens, Athens 15780, Greece

Corresponding author: Nikolaos Nikolaidis (e-mail: it14189@uom.edu.gr).

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 801015 - EXA2PRO (https://exa2pro.eu)

ABSTRACT Technical Debt (TD) is a software engineering metaphor that resembles the production of poor-quality code to going into debt. In particular, a development team that "*saves*" effort while developing by not removing inefficiencies, has to "*pay-back*" with interest, in the form of additional maintenance costs (i.e., fixing bugs, adding features, etc.) due to the poor maintainability of the developed code. Although maintainability assurance is an established practice in traditional software development (lately known as TD management), it has still not attracted the attention of scientific software developers; i.e., researchers writing code and developing tools for purely research purposes. Nevertheless, based on the literature and practice, maintainability seems to be ranked as an important key-driver for the development of such applications; since the effort needed to update the code before the experimentation (e.g., executing a simulation) is common and should not receive low priority. In this paper, we present the outcome of a 3-year research project on Technical Debt Management (TDM) for scientific software development. The outcome of the project is a framework (termed: EXA2PRO TDM framework) and an accompanying platform for assisting scientific software developers in managing the TD of their applications. The framework is a collection of methods tailored for the mainstream programming languages of scientific software development, which have been empirically validated through five pilot applications. The majority of the EXA2PRO framework suggestions have been applied by scientific software developers and eased future maintenance activities.

INDEX TERMS code quality, refactoring, scientific software development, and technical debt

I. INTRODUCTION

Scientific software development refers to the process of developing software applications for research purposes (e.g., simulations, large-scale data analytics, etc.) [1]. The execution of such applications is so time-consuming that they are usually executed on High-Performance Computing (HPC) infrastructures [2]. The long execution time of scientific software applications can lead to substantial "*loss of resources*" if execution of the software fails; rendering maintenance activities (such as bug-fixing, updating an algorithm, etc.) of paramount importance both in terms of correctness and efficiency. We note that maintainability has been highlighted as comparably important to performance and scalability in the field of software engineering for scientific computing, based on a recent secondary study [3].

To assure the maintainability of software systems, in "*traditional*" software engineering, the concept of Technical Debt Management (TDM) has been adopted along the last decade [4] as a means of highlighting, in monetary terms, the maintainability problems that should be fixed as well as the associated effort for fixing them. Technical Debt resembles the deterioration of maintainability to going into debt: the effort that a company saves (termed as *principal*) while developing a software in a suboptimal maintainability state is paying *interest*, in the form of additional effort needed to maintain the software (compared to the effort that would be required if the software was of optimal maintainability) [5]. To bridge the two communities (the scientific software development and the TD community), the EXA2PRO research project (exa2pro.eu), among other goals, attempts to

bring knowledge and best practices from the software engineering community (which is more knowledgeable in developing software) to scientific software development (which urges for applying those practices). To achieve this goal, the project delivers the EXA2PRO TDM framework, which tailors TDM methods and tools to fit the scientific software development domain, e.g., given programming languages and the imposed run-time constraints—need for high levels of performance (in terms of time), interoperability, hardware heterogeneity, etc.

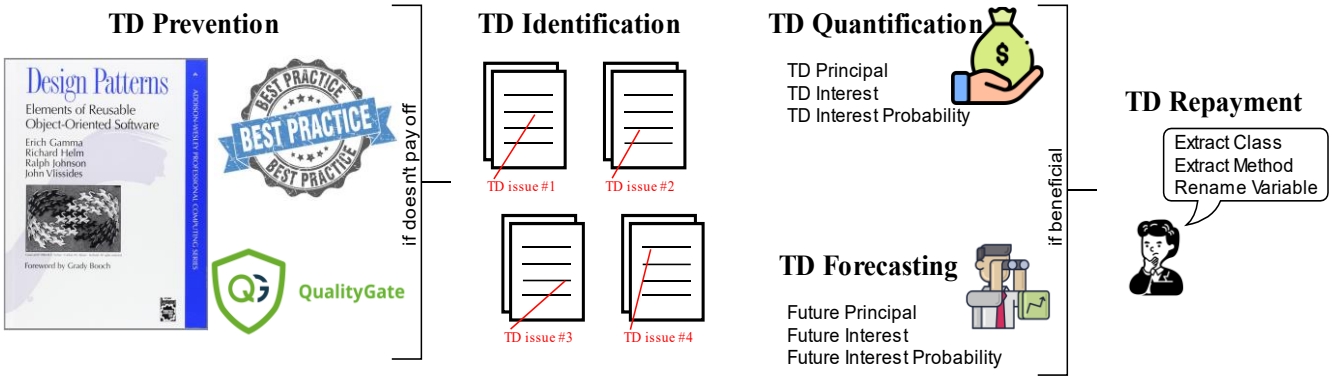

FIGURE 1. EXA2PRO TDM Framework: Bird's Eye View

In Figure 1, we present the high-level view of the EXA2PRO TDM framework. In particular, the goal of the project is to cultivate a culture of *TD Prevention*, i.e., that new TD is not introduced into the system. However, this is not fully feasible in practice [6]. Therefore, if technical debt prevention does not pay off, some technical debt items will eventually creep into the system, or they might exist in legacy code (the code that pre-existed the adoption of the EXA2PRO TDM framework). By analyzing the code, the framework will provide developers a list of items (files, procedures, etc.) that suffer from TD (*TD Identification*). Since the number of these items is expected to be quite high, a *TD Prioritization* approach that ranks them in terms of urgency to resolve is required. For EXA2PRO TDM framework, the prioritization relies on the outcome of two systematic activities, namely *TD Quantification* (that assesses the current values of Principal and Interest) and *TD Forecasting* (that predicts the future values of these TD aspects). Next, being supported by the prioritization process, the software engineers must decide which TD items (files, procedures, etc.) they should focus on, and apply targeted refactorings (*TD Repayment*), gaining maintainability.

In this paper, we focus on TD identification, quantification, and repayment. From all the ***TD identification*** approaches in the literature (e.g., static-code analysis, self-admitted TD, etc.), the one used by EXA2PRO framework relies on a metric-based approach that flags files and procedures (or modules in FORTRAN) with extreme metric scores for the complexity, coupling, and cohesion quality properties. These files / procedures urge for refactoring, since they are expected to hinder future maintenance. In addition to that, we assess the time that would be needed for the manual resolution of each type of problem, approximating Principal, based on the type of problem (***TD quantification***). In terms of refactoring these problems (***TD repayment***), we have updated an existing approach for the decomposition of Long Procedures (SEMI [7]), so that it scales for extremely long artifacts and adapted the Agglomerative Clustering Technique, proposed by Fokaefs et al. [8] to decompose Large Files / Modules into more coherent ones. The proposed refactorings are the Extract Procedure and the Extract File / Module that are able to improve multiple code qualities: namely, decrease size, complexity, coupling and increase cohesion. To validate the EXA2PRO TDM framework, we have assessed the usefulness of the proposed methods and tools on five (5) real-world scientific software applications, from the pilot case providers of EXA2PRO project (i.e., CERTH[1], Julich[2] and CNRS[3]). In terms of programming languages, we focus on FORTRAN and C, since these languages are heavily used in HPC software [3].

The rest of the paper is organized as follows: in Section II we present background information and related work in terms of technical debt management and scientific software development. In Section III, we present in detail the novel approach for TD identification and quantification; in Section IV we present an overview of how we adapted the refactoring approaches; and in Section V we illustrate the provided tool. In Section VI we provide the empirical results on the usage of the EXA2PRO TDM framework. Finally, we conclude the paper and present threats to validity in Section VII.

II. BACKGROUND INFORMATION

A. TECHNICAL DEBT MANAGEMENT

The technical debt (TD) metaphor was introduced in 1992 by Ward Cunningham [9]. Cunningham used this analogy to

[1] https://www.cperi.certh.gr/en/
[2] https://www.fz-juelich.de/ias/jsc/EN/Home/home_node.html
[3] https://www.cnrs.fr/

emphasize the consequences of shipping "*not-quite right*" code (code with inefficiencies) for the first time. The presence of these inefficiencies hinders future software maintenance acting as interest that needs to be paid [9]. In the next decades, this metaphor gained a lot of ground and is currently considered as an established terminology in both academia and industry. The primary benefit of the TD metaphor is that it serves as a channel of communication between technical and non-technical stakeholders [10].

1) TECHNICAL DEBT CONCEPTS

While applying the metaphor, the software engineering community has borrowed the concepts of principal and interest from economics. In the context of TD, *Principal* is the effort required from the developers to remove code and design inefficiencies; thereby, bringing the software closer to an optimal quality. Although, we acknowledge that the notion of optimal quality might be utopic for software, for the sake of applying the metaphor, the community considers as optimal a hypothetical version of the software system under consideration, with improved maintainability. On the other hand, *Interest* is the extra effort required to maintain the software, in comparison to the effort that would be needed if the system was in the optimal state [5].

For the calculation of the TD Principal several automated tools have been developed, but SonarQube is one of the most widely used [11]. SonarQube calculates TD Principal by identifying fragments of code that violate certain predefined rules and associates these violations (TD issues) with the time required to resolve them. We note that, since SonarQube principal calculation relies on micro level coding violation and does not considers more prolific issues (such as the violations of design principles, etc.), in this paper we do not rely on SonarQube for the calculation of principal, but describe a novel approach—see Section III.

While principal can be calculated in a straightforward manner when referring to code TD (using static analyzers), the calculation of TD Interest is far more challenging, as it assumes the knowledge of an 'optimal' system as well as the difference of the actual system to that optimal. Conejero et al. [12] found that maintainability is one of the main contributors of TD Interest and Seaman and Guo [13] established a similar relationship. These studies paved the way for the usage of proxies for the estimation of TD Interest. In EXA2PRO, the calculation of interest relies on the FITTED framework (Framework for Managing Interest in Technical Debt) introduced by Ampatzoglou et al. [14]. The calculation of the TD interest relies on: (a) identifying comparable software artifacts so as to judge optimality among structurally similar artifacts, (b) constructing a hypothetical optimal artifact as a collection of maintainability scores of all similar artifacts; (c) calculating the distance of the artifact under consideration from the hypothetical optimal; and (d) monetizing the effort required to perform a future change, based on the distance and past maintenance effort on the specific artifact—more details are provided in [15].

2) TECHNICAL DEBT MANAGEMENT

According to Li et al. [4] the management of TD consists of eight activities: repayment (i.e., reducing the accumulated TD), identification (i.e., finding artifacts with excessive TD values), measurement (i.e., quantifying TD), monitoring (i.e., recording and valuation of TD evolution), prioritization (i.e., find items that needs to be repaid first), communication (i.e., explain TD to stakeholders), prevention (i.e., keep away of additional TD), representation / documentation (i.e., record metrics, actions about TD).

According to Eisenberg [6], the complete repayment of TD is not a realistic goal. In particular, the current literature supports that it might be profitable to prioritize the repayment of TD in parts of the code, which are rarely the subject of maintenance activities [13]. Based on the above, continuous management of TD is required, so as to consider not only software quality, but also the effort required to make changes, and the cost of investment on software improvement [10].

B. SCIENTIFIC SOFTWARE DEVELOPEMENT

A literature review by Heaton and Carver [16] shed light on how the scientific software development community is using software engineering practices (we note that the vast majority of scientific applications are executed in HPC infrastructures). They found that "*Issue Tracking*" and "*Version Control Systems*" are the most adopted practices, but there is still room for improvement. Another study found similar results with validation and testing being the least adopted ones [17]. Moreover, the literature review of Sletholt et al. [18] focused on the effect of the agile practices being used in HPC. The results of this study showed that agile practices achieve better testing and requirement results. Based on these studies, there is evidence that HPC developers care about software technology practices, as they seem to have a positive impact on the development of software.

To empirically assess the EXA2PRO TDM framework, in this study, we used five HPC software applications, provided by three pilot providers of the EXA2PRO project. CERTH provided two versions of the CO_2Capture application, which is a simulator of the design and control of chemical processes and materials in CO_2 capture. CNRS contributed through the MetalWalls application, which accurately simulates the behavior of supercapacitors. Finally, JULICH provided the LQCD and KKRnano applications, which implement the functionality of the Grid LQCD library and the core operation of the density functional theory.

III. EXA2PRO TD IDENTIFICATION / QUANTIFICATION

In this section, we present our approach for identifying artifacts that suffer from technical debt and quantify their TD Principal at the design level. Design Debt is calculated as the amount of money corresponding to the effort required to resolve design inefficiencies [4]. To quantify Design Debt Principal, in the EXA2PRO TDM framework, we have followed a 4-step approach:

1. define a list of design problems to be identified
2. identify items (i.e., files, modules, or procedures) that suffer from these design problems
3. estimate the time required to fix each design problem
4. sum the time required to fix all identified design problems, in all items

Each one of the aforementioned steps is detailed in the upcoming sub-sections. We note that Design Debt Interest calculation follows without any deviation the FITTED framework [14]; therefore, we exclude it from this paper.

A. DEFINITION OF DESIGN PROBLEMS

As a starting point for identifying tentative design problems, to be captured by the EXA2PRO TDM framework, we used the seminal book on refactorings by Fowler and Beck [19]. After examining the design problems presented in the book, and by considering the fact that in scientific software applications, the use of object-orientation is sparse, we decided to focus on four design problems that can also fit in the imperative and procedural programming paradigms:

- ***complex artifacts***: the body of some procedures presents excessive levels of complexity, in terms of decision or iteration nodes. Such code chunks (and files containing them) are difficult to understand and maintain [20].
- ***over-coupled artifacts***: some files or modules depend on an excessive number of external files, since they need their information to compile. Such files are prone to ripple effects, i.e., they need retesting every time that a dependent file changes and are also hard to reuse. This leads to additional maintenance effort [21].
- ***large artifacts***: some artifacts (modules, files, or procedures) are of large size (usually in terms of lines of code). These artifacts (procedures) have more reasons to change (i.e., due to their additional responsibilities). These artifacts violate [22] the Single Responsibility Principle (SRP) [23]; thus, are more probable to undergo maintenance and produce TD Interest.

We note that the above list is by no means comprehensive; thus, the captured Design Debt Principal will be a fraction of the actual one. However, we consider this list as appropriate, at least as a starting point, since: (a) it captures the most important (non-object-oriented) properties of software maintainability [24] and (b) it would not be feasible to capture all types of design problems in the course of the project. Summarizing the above, and by considering that artifacts in non-object-oriented languages are usually files and procedures, the following design problems need to be treated: (a) ***Complex Procedures***, (b) ***Over Coupled Files/Modules***; (c) ***Large Files/Modules***; and (d) ***Long Procedures***.

B. IDENTIFICATION OF DESIGN PROBLEMS

To identify design problems in large code-bases a scalable approach that can be automated is required. To this end, we have opted for a metric-based approach for identifying problems [25], i.e., to calculate the values of suitable metrics for each type of problem, sort the artifacts (for the case of EXA2PRO framework: files and procedures) in terms of each metric, and mark the worse ones, as problematic. The approach of using metric thresholds as indicators of problematic artifacts is well-cited in the literature [26]. In the next subsection, we present the metrics' selection process, as well as details on their calculation; while after that, we present the approach for extracting the metric thresholds and the actual values that we have retrieved.

C. METRICS SELECTION AND CALCULATION

As a first step towards the application of the proposed methodology, we need to select the metrics that we will use for the identification of design problems. This strategy (i.e., using metrics to identify design problems) is well-established in the software engineering literature [27]. Based on the problems that we have defined in Section III.A and the quality properties considered for applying a "good design" paradigm (i.e., low coupling, high cohesion, and low complexity) [27], the EXA2PRO TDM framework calculates five metrics:

- cyclomatic complexity (CC) [20]—for identifying ***Complex Procedures***;
- coupling between files (CBF)—for identifying ***Over Coupled Files*** / ***Modules***;
- lines of code (LOC) [28]—for identifying ***Large Files*** / ***Modules***;
- lack of cohesion of lines (LCOL) [7]—for identifying ***Long Procedure***; and
- lack of cohesion of procedures (LCOP)—for identifying ***Large Files***/***Modules***.

From the above list, three metrics (namely: CC, LOC, and LCOL) are reused, as they have been proposed in the literature; whereas the other two are introduced (CBF and LCOP) as part of this paper. Nevertheless, we need to note that CBF and LCOP are not developed from scratch, since they rely on Coupling between Objects (CBO) and Lack of Cohesion of Methods (LCOM) [22]. In particular: (a) CBF refers to the number of external dependencies of files/modules; whereas (b) LCOP refers to the number of disjoint procedures in terms of variables' usage. To be able to calculate the aforementioned metrics, we need to differentiate between FORTRAN and C, in the sense that they have a completely different approach for managing the scope of variables, which directly affects how coupling and cohesion is perceived/defined in the two languages. The differences in the calculation of these metrics between the two languages (e.g., treating global variables) are presented in detail in Appendix A and Appendix B, respectively.

To summarize the above, in Table I, we provide an overview of calculated metrics per design problem. For cases in which more than one metrics are used for identifying the existence of a specific design problem (i.e., Large Files / Modules), a union of the artifacts identified by the metrics, is performed. Finally, we note that `modules` are applicable only

to FORTRAN 90 and the LCOP metric is not applicable to FORTRAN 77.

TABLE I: *METRICS SELECTION OVERVIEW*

Design Problem	CC	CBF	LOC	LCOL	LCOP
Complex Procedures	X				
Long Procedures				X	
Over Coupled Files / Modules		X			
Large Files / Modules			X		X

D. THRESHOLDS IDENTIFICATION

Given the fact that the projects of the used code-bases are highly divergent in terms of size, required complexity, etc., we have preferred to set project-specific thresholds, rather than global ones, noted as more appropriate in the study of Mori et al. [29]. This decision relies on the fact that regardless of how '*good*' or '*bad*' the quality of a code-base is, the refactoring budget is limited, and cannot spread to a large (or the complete) number of artifacts. The thresholds for each one of the pilot cases (identified at the 10% of worst artifacts, per metric) are presented in Table II. Similar cut-off percentiles have been used in other studies aiming at the derivation of metric thresholds [30]. We note that both versions of CO_2Capture are evaluated using the same threshold values.

TABLE II: *PROJECT-SPECIFIC THRESHOLDS*

Project	CC	CBF	LOC	LCOL	LCOP
LQCD	6	3	30	193	26
KKRnano	1	4	24	2283	36
Metalwalls	7	11	75	1309	1
CO_2Capture	2	5	49	1358	78

E. VALUATION OF SOLVING DESIGN PROBLEMS

As a solution to the problems defined in Section III.A, we propose the application of two well-known refactorings: *Extract Procedure* and the *Extract File / Module*. In particular, *Extract Procedure* targets the *Long Procedure* and the *Complex Procedure* design problems by moving a code fragment to a new method / function and replacing the old code with a call to the new method. The Extract Procedure (Method) refactoring is the most common type of refactoring according to a study of 16,566 identified refactorings in the version history of 23 projects [31]. On the other hand, the *Extract File / Module* refactoring is expected to resolve the *Large File / Module* and the *Over-Coupled File / Module* design problems by creating a new File / Module and place the fields and methods for the relevant functionality in it. The Extract File / Module refactoring (class) in object-oriented systems is considered as one of the more global ones [32].

To this end, the goal of this subsection is to estimate the time needed to perform these refactorings, without any tool support; so as to assess the time that is needed to eliminate one occurrence of the design problem. To achieve this goal, we worked on the code-bases of two pilot applications: Metalwalls (developed in C) and CO_2Capture (developed in FORTRAN). To systemize the process of refactoring effort valuation, we applied the following process:

- retrieve artifacts that suffer from design problems
- design a solution for solving the problem—record mental process effort (in minutes)
- apply the solution in the code-base (including re-testing)—record the actual implementation effort (in minutes).
- multiply the sum of the two calculated effort values, with the average salary of the developer (per minute)

We note that (a) the procedures of the 2nd and 3rd steps have already been performed and only for extreme precision purposes they should be tailored to the companies' specifications, and (b) the 4th step is performed based on a global average of developers' hourly rate, but it can be tailored to map any salary cost of specific companies or countries.

Valuation of Extract Procedure Refactoring. Regarding the *Extract Procedure* refactoring, we have manually identified 84 opportunities in the code-base of Metalwalls and 47 on the code-base of CO_2Capture. To explore the time to apply the extract refactoring procedure, we focus on a single file, namely the `System.F90` file, which includes 21 extract procedure opportunities. In Table III, we present the effort required to fix each instance of the long procedures.

TABLE III: *TD PRINCIPAL ANALYSIS FOR SYSTEM.F90*

Source File	Extract Procedure	Time to Resolve (Minutes)
read_data()	open_file()	6
	read_header()	8
	check_if_value_set()	10
	check_all_keywords()	6
	validate_values()	8
	read_box_parameters()	5
	write_box_lengthparameters()	10
	validate_array_coordinates_dimensions()	15
	read_coordinates()	5
	read_ions()	13
	read_atoms	9
	validate_array_velocities()	7
	read_velocities()	8
	read_ions_velocities()	6
	check_atoms_velocities()	11
	read_forces()	16
	read_thermostat_parameters()	8
	read_electrode_atom_charges()	10
	finilize_system_setup()	18
deallocate _data_arrays()	perform_deallocation()	23
	setup_do_output()	24

An example of such a code transformation is presented in Figure 2: on the top side of the figure, we present the code before the application of the refactoring, whereas on the bottom side the source code after. Since this particular piece

of code (on the top) was performing a specific procedure (reading a data file from the system,) we decided to perform an *Extract Procedure* refactoring, by creating a procedure that reads a data file from the system. Later, we generalized the use of this procedure to read a file from the system (either config, or data file) and we transferred it to the `fileunit.F90` source file.

```
! Open file
call MW_fileunit_get_new_unit(funit)
open(unit=funit, file=datafile,  &
     access="SEQUENTIAL", action="READ",  &
     position="REWIND", form="FORMATTED",  &
     status="OLD", iostat=ierr)
if (ierr /= 0) then
     call MW_errors_open_error("read_data",  &
          "configuration.f90", datafile, ierr)
end if
```

```
! Open file
call MW_fileunit_open_file(funit, ierr,  &
     datafile)
```

FIGURE 2. ***Example Application of Extract Procedure Refactoring.***

Next, we replicated the aforementioned process in the 47 opportunities of the CO_2Capture project. The statistical analysis (on the complete dataset) for the valuation of TD Principal for refactoring Long Procedures suggests that on average, each instance requires 9.89 minutes to be refactored. The minimum value is 1 minute, the maximum value is 48, whereas the standard deviation is 8.63. The descriptive analysis is visualized in the boxplot of Figure 3. The analysis for rest instances is presented in Appendices C-E.

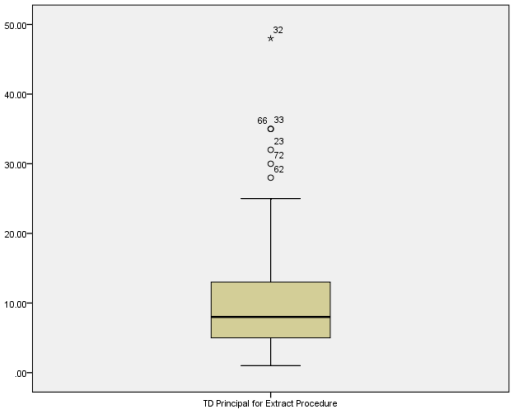

FIGURE 3. ***TD Principal for Extract Procedure***

Valuation of Extract File / Module Refactoring. Regarding the Extract File / Module refactoring, we have manually identified 5 opportunities in the code-base of Metalwalls and 6 on the code-base of CO_2Capture. The statistical analysis on the valuation of TD Principal for refactoring Large Files / Modules suggests that on average, each instance requires 19.20 minutes to be refactored. The minimum value is 14 minutes, the maximum value is 28, whereas the standard deviation is 5.54, see Figure 4.

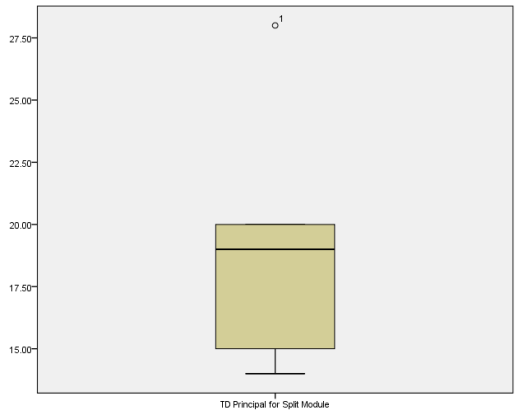

FIGURE 4. **TD Principal for Extract File/Module**

H. FINAL ASSESSMENT OF DESIGN DEBT PRINCIPAL

The final step of this process is straightforward in the sense that it corresponds to the calculation of a weighted sum of the occurrences of each design problem multiplied by the cost to resolve each problem. To synthesize the results of the two projects in a common formula, we first examine if there is a statistically significant difference in the mean time required to fix each design problem in the two projects. For both cases (as it is also visually inspected by contrasting the boxplots—in pairs), the differences in the two projects are not statistically significant. Therefore, as a remediation time for each design problem resolution, we use the average value of the joined dataset from the two projects (9.98 minutes for the Extract Procedure refactoring and 19.20 for the Extract File / Module refactoring). To transform the effort required in minutes to currency (i.e., euros) we use the average monthly rate of the three pilot case providers (i.e., 39.44 euros per hour). Thus, Technical Debt Design (TDD) principal can be calculated as follows (in euros), taking into account that the cost for applying the Extract Procedure refactoring is 6.56 euros, whereas the cost for applying the Extract File / Module refactoring is 12.62 euros:

$$TDD_{Principal} = \left(\#_{long\ procedure} + \#_{complex\ procedure}\right) * cost_{extract\ procedure} + \left(\#_{large\ file/module} + \#_{overcouple\ file/module}\right) * cost_{extract_{file/module}}$$

$$= \left(\#_{long\ procedure} + \#_{complex\ proceduce}\right) * 6.56 + \left(\#_{large\ file/module} + \#_{overcouple\ file/module}\right) * 12.62$$

IV. EXA2PRO TD REPAYMENT

In this section we present the employed approaches for the automated identification of *Extract Procedure* and the *Extract File/Module* refactoring opportunities. The two approaches are tailored versions of the approaches originally presented by Charalampidou et al. [7] and Fokaefs et al. [8]; thus, they are presented in brief.

Applying the Single Responsibility Principle for Extracting Procedures. The approach that we use for splitting a long procedure relies on the Single Responsibility Principle (SRP)

[23], which was inspired by the functional module decomposition, introduced by De Marco [33]. In particular, we relied on the way that the SRP has been applied by Charalampidou et al. [7], for proposing the SRP-based Extract Method Identification (SEMI) approach. The approach utilized the relation between fragments of code that collaborate to complete a functionality, by assessing the cohesion among them (i.e., using same variable or calling the same method). Based on the above, SEMI identifies all possible coherent sets of successive statements, by following the process shown in Figure 5.

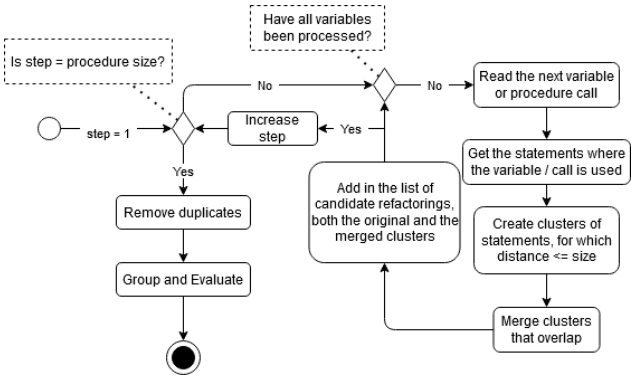

FIGURE 5. Flow chart of Extract Procedure opportunities

Decomposing Files Using an Agglomerative Clustering Technique. The clustering algorithm that we use for the file / module decomposition is the *agglomerative algorithm*, a type of *Hierarchical Clustering*. In general, *Hierarchical Clustering* seeks to build a hierarchy of clusters and is based on the core idea of placing entities being more related to nearby entities than to entities farther away. As such, these algorithms connect entities to form clusters based on their distance. A cluster can be described largely by the maximum distance needed to connect parts of the cluster. The *Agglomerative Clustering* algorithm can be outlined as follows: At the initialization step, it assigns each entity to a single cluster. In each iteration, it merges the two clusters with the minimum distance. The algorithm terminates when all entities are contained in a single cluster. To be able to decide the actual clusters, we must select a threshold value for the minimum distance as a cut-off value. The hierarchy of the clusters is usually represented by a dendro-gram. The leaves of the tree represent the entities, the root is the final cluster and the intermediate nodes are the actual clusters. The height of the tree represents the different levels of the distance threshold in which two clusters were merged.

There are plenty of methods to select the closest clusters. We chose the *Average Linkage* method, in which the distance between one cluster and another one is considered to be equal to the average distance from any member of one cluster to any member of the other cluster. As for the threshold (cut-off) value for the minimum distance, we do not define a fixed one, but we apply the agglomerative clustering algorithm for a range of threshold values (from 0.1 to 1.0) and we present the results. We have observed that higher thresholds (ranging from 0.85 to 1.0) generally produce better results than lower ones. The distance metric we chose to use is the *Jaccard Distance*, which produces decent results in software re-modularization. To define the *Jaccard Distance* between two procedures, we use the notion of *entity sets.* According to this notion, the entity set of a procedure contains all procedures (subroutines & functions) that are invoked by the procedure, all attributes that are accessed by it and the procedure itself. Thus, having defined the notion of *entity sets*, we calculate the *Jaccard Distance* between two entity sets *A* and *B.*

V. TOOL SUPPORT

The EXA2PRO TDM toolbox is released both as an Eclipse plugin[4] and as a standalone[5] application. The main functionalities of the EXA2PRO TDM toolbox (plugin version) are presented below.

New-Load-Delete Analysis: The user of the plugin is able to start a new analysis, load the last analysis, and delete the analysis of a project. These options are available from the corresponding toolbar icons and the project popup menu.

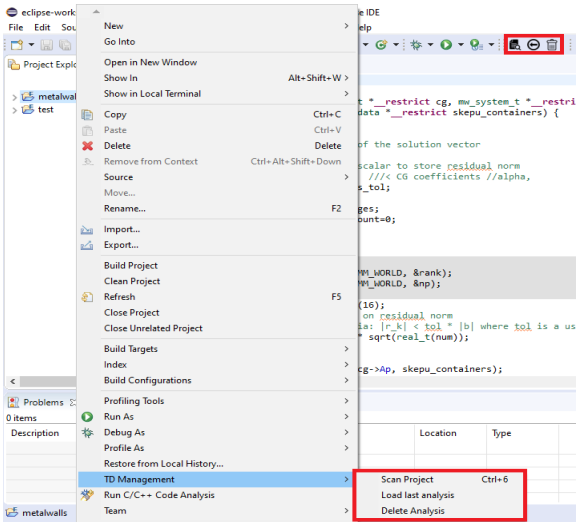

FIGURE 6. The extra toolbar icons and menu options.

Metrics View: The Metrics view is a table where all the calculated metrics (Fan-Out, Cohesion, Cyclomatic Complexity, and Lines of Code) are presented. In this view, there are two options to show the file metrics or the method/function metrics. The user can change the option through the menu of this view or by using the icons in the toolbar, for file and method/function metrics respectively.

[4] https://github.com/nikosnikolaidis/Exa2Pro-Plugin

[5] https://github.com/nikosnikolaidis/Exa2Pro

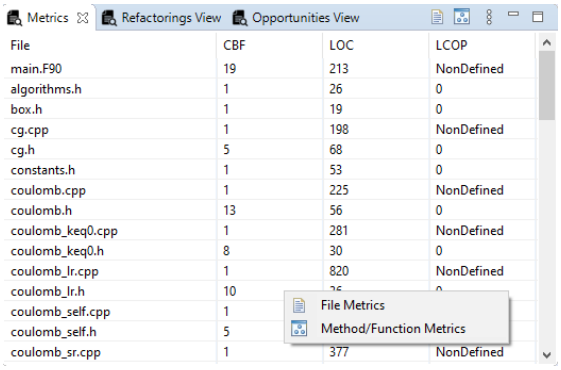

FIGURE 7. The Metrics view.

Refactorings View: In the Refactorings view, all files and methods/functions in need of some kind of refactoring because of an excessive metric value are displayed. In this view, there are four different options, one for each metric. Moreover, the user can start the TD repayment process from here, one can select a procedure or file / module in order to start the process of finding specific opportunities for refactorings.

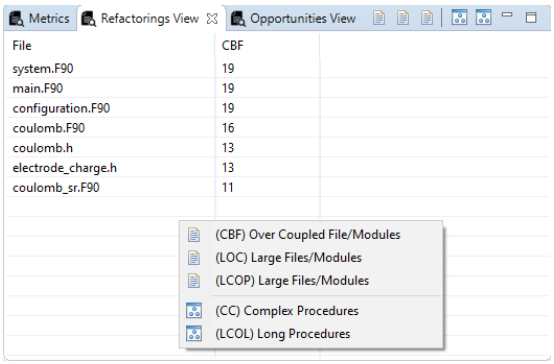

FIGURE 8. The Refactorings view.

Opportunities View: Once a procedure or file / module from the refactoring view has been selected and the analysis process is finished, the Opportunity view is populated with all the possible refactorings. These refactorings are opportunities for extracting methods from the designated lines yielding the shown benefit in terms of method cohesion. For convenience, if an opportunity is chosen by the user the specific file opens with the corresponding lines already selected.

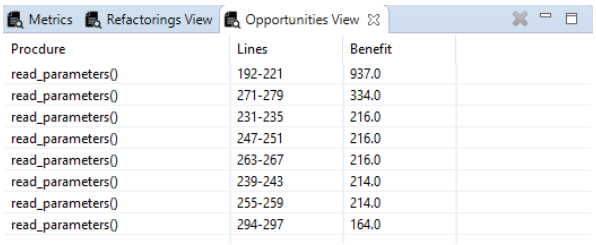

FIGURE 9. The Opportunity view.

In addition to the above, the plugin offers a *chart view* to visualise the evolution of metrics, a *markers' view* to see the suggestions as warnings / errors in the Eclipse IDE, *preferences*, and *help*.

VI. EMPIRICAL RESULTS

In this section, we present the results of using the proposed framework for TDM on the pilot applications of EXA2PRO. We note that projects CO2Capture-1 and CO2Capture-2 are different versions of the same project; however, they differ substantially as the 2nd version adopted several performance optimizations. For each project we record the following:

- number of identified design problems, TD Principal, and TD Interest;
- Applied refactoring opportunities;
- assessment of refactoring opportunities in two ways: conceptual assessment (fitness of refactoring) and TD assessment (design TD and TD Interest)

The results are organized into three subsections, based on the steps followed to locate and mitigate inefficiencies.

A. MEASUREMENT AND IDENTIFICATION

The first step for each of the cases refers to the measurement and identification process as described in Section III. Table IV depicts the number of Design Debt issues that have been identified in each case, along with the design-level TD Principal and TD Interest in monetary terms (euros).

TABLE IV: *PROJECTS' TD IDENTIFICATION*

Case	Design Debt issues	Design TD	TD Interest
CO_2Capture-1	51	329.00	664.94
CO_2Capture-2	60	447.10	1,694.58
MetalWalls	71	474.80	776.22
LQCD	15	106.99	103.53
KKRnano	93	636.85	685.74

B. APPLIED REFACTORING OPPORTUNITIES

Upon identification, the developer is aware of the artifacts that suffer from excessive metric scores and constitute candidates for refactorings application. For each project, we applied the Extract Procedure and Extract File / Module refactorings, prioritized based on the metric scores. We note that due to limitation of resources we have chosen not to fix all identified issues. In Table V we present the number of applied refactorings for each project. We should note that the LQCD project exhibits fewer opportunities as it is much smaller in size than the rest.

TABLE V: *PROJECTS' APPLIED REFACTORING*

Case	Extract Procedure	Extract File/Module
CO_2Capture-1	25	1
CO_2Capture-2	39	6
MetalWalls	79	5
LQCD	1	-
KKRnano	7	1

C. ASSESSMENT OF APPLIED REFACTORINGS

After the application of the selected refactorings, we conducted short interviews with the developers of the projects (along with a questionnaire) for assessing the changes. Based on collected data, we were able to assess the conceptual and structural fitness of our refactorings (i.e., the extent to which they made sense to the developers). Finally, we quantitatively analysed the effect of the changes on TD Interest. Acknowledging that performance is a non-negotiable priority in scientific software applications, before proceeding with the presentation of TD results, we note that the proposed changes have not drastically affected the performance. The aggregate impact of all applied refactorings on the performance (percentage change in execution time) of each project is visible in Table VI. The changes to the execution time were considered acceptable by the developers.

TABLE VI: *IMPACT OF REFACTORING ON PERFORMANCE*

Case	Impact
CO_2Capture-1	- 0.4%
CO_2Capture-2	+ 0.5%
MetalWalls	- 0.25%
LQCD	~0%
KKRnano	-

From the total number of refactoring opportunities identified in the projects (224), only 9 of them were noted as being not conceptually correct. For the rest of the unaccepted refactorings, the developers would prefer the code in its original form or in an alternative format, without however stating that the refactoring was flawed. The total accepted refactorings rate, for adoption in the final source code, is presented for each project in Table VII.

TABLE VII: *PROJECTS' ADOPTED REFACTORINGS*

Case	Adopted Extract Procedure	Adopted Extract File/Module
CO_2Capture-1	25/25	1/1
CO_2Capture-2	36/39	6/6
MetalWalls	61/79	2/5
LQCD	0/1	-
KKRnano	0/7	1/1

In Table VIII, for each project, we present the change of TD Interest as a percentage. A negative percentage accounts to a reduction in the metric score (i.e., improvement of quality for all metrics), while a positive percentage refers to an increase of the metric score (i.e., a deterioration of quality). Next, we present a qualitative assessment of the refactoring procedure through quotes captured during the interviews with developers.

TABLE VIII: *PERCENTAGE OF CHANGE IN METRICS AND TD INTEREST DUE TO APPLIED REFACTORINGS*

Case	CC	LCOL	LOC	CBF	LCOP	TD Interest
CO_2Capture-1	-10.3	-60.9	0.5	0.0	0.0	-19.0
CO_2Capture-2	-38.0	-80.7	-10.4	-5.3	-26.1	-21.9
MetalWalls	-16.0	-71.4	-0.4	2.0	-75.0	-31.5
LQCD	-9.1	-11.0	48.8	-7.1	-10.0	0.5
KKRnano	0.0	-7.2	-4.6	10.0	-0.1	-4.9

CO2Capture. First of all, we should note that during the interview the developers explicitly mentioned that "*This is a general code base that we use for multiple projects, so these refactorings are very beneficial*", implying an even greater impact on the maintainability of the affected systems. The application of the refactorings led to a reduction in metric values. The change is more striking for the LCOL metric, but it is also significant for the CC metric as well as the TD Interest. The lines of code were slightly increased, as a result of extracting code to separate procedures, which is reasonable for this type of refactoring. During the 2nd meeting with the developers (2nd round of the refactoring process), it became evident that they were quite interested in the potential of the applied refactorings, but caution should be exercised so as to not hurt the performance. As it can be observed in Table VII, all of the system metrics along with the TD Interest experienced a significant improvement (decrease). The decrease was higher in this project since we had the opportunity of applying more refactorings of both types (Extract File / Module and Extract Procedure).

MetalWalls. During the interview with the developers of this application, regarding the refactorings that were accepted, it was brought up that "*These kinds of contributions make perfect sense and it can be even pushed to the production code on the spot*". On the other hand, for the refactorings that were not adopted the developers noted that "*Some Extract File / Modules make less sense because it is more practical to change only one file (in the future), rather than searching in multiple ones, but this is more like a habit in the HPC community*". For this project, we can see a similar improvement, again due to the large number of the applied refactorings. A deterioration was observed only for the CBF metric (by 2%) which is due to Extract Files / Modules refactorings introducing additional dependencies between files/modules.

For the *LQCD* and *KKRnano* project, the developers have not accepted the majority of the proposed refactorings because of their programming style, as they observed that "*We wouldn't apply these changes as they don't fit the programming style of the specific domains*". The LQCD application is quite small (compared to the rest cases) and we were able to apply only one refactoring. At a first glance, for the corresponding developer, the recommended refactoring has not been very appealing, due to the separation of comments in the code.

Regarding the changes to the metrics, all of them have been improved, apart from LOC which increased because of the extra lines required to call and initialize the new procedure. Finally, for KKRnano, almost all metrics have been improved or remained stable (apart from the value of CBF, which increased as a result of the introduction of new files/modules). It is worth mentioning that the TD Interest presented a non-negligible improvement as well.

VII. CONCLUSION

In this paper we presented a Technical Debt Management (TDM) framework that supports the quality assurance of scientific software applications. The paper details:

- ***a TD quantification at the design level***. In particular, long and non-cohesive files and procedures which are in need of refactoring are identified through excessive metric values. Furthermore, the TD principal associated with each type of design problem has been estimated.
- ***two refactoring techniques for addressing the aforementioned design problems***. In particular, we updated the SEMI approach for the decomposition of *Long Procedures* and we adapted the Agglomerative Clustering Technique to decompose *Large Files / Modules* into more coherent ones.
- ***a developed tool*** (implemented both as a standalone tool and in the form of an eclipse plugin)
- ***empirical evidence*** on the *TD Interest* benefits that are obtained by applying *TD Repayment*.

The exploratory application of the proposed design-level TD refactorings on five HPC software projects revealed that maintainability can be substantially improved in scientific software applications. For example, the refactorings applied on the studied applications have reduced TD interest by 21.9%, 31.5% and 4.8%, respectively. At the same time, the application of these refactorings on the performance of the corresponding applications was rather minimal, ranging from a 0.4% improvement to a 0.5% deterioration in the execution time, depending on the refactorings that have been applied. Thus, there is sufficient evidence to support the claim that design and code level improvements on the code-base of scientific software applications can increase their level of maintainability without harming their performance. Furthermore, we need to acknowledge that despite the expected difficulties from the scientific software developers to understand all the details of the EXA2PRO framework (i.e. the notion of the TD principal and interest, design problems, and metrics selection), the simplified version offered through the tool will ease the adoption of the proposed approach. Moreover, as interesting future work directions we highlight the exploitation of other TD identification methods, such as the presence of self-admitted technical debt (SADT), or analysis of other types of artifacts (e.g., architectural TD). Finally, we believe that an additional interesting future work direction will be the fine-grained assessment of the effect of the aforementioned refactorings on performance.

REFERENCES

[1] C. K. Birdsall and A. B. Langdon, "Plasma Physics via Computer Simulation", the Adam Hilger Series on Plasma Physics. Adam Hilger, New York, 1991.

[2] M. Schmidberger and B. Brügge, "Need of Software Engineering Methods for High Performance Computing Applications", 11th International Symposium on Parallel and Distributed Compu-ting, Munich, Germany, 25-29 June 2012.

[3] E.-M. Arvanitou, A. Ampatzoglou, N. Nikolaidis, A.-A. Tzintzira, A. Ampatzoglou, and A. Chatzigeorgiou, "Investigating Trade-offs between Portability, Performance and Maintainability in Exascale Systems," in 2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Aug. 2020, pp. 59–63.

[4] Z. Li, P. Avgeriou and P. Liang, "A systematic mapping study on technical debt and its management", Journal of Systems and Software, vol. 101, pp. 193-220, 2015.

[5] A. Chatzigeorgiou, A. Ampatzoglou, A. Ampatzoglou and T. Amanatidis, "Estimating the breaking point for technical debt," 2015 IEEE 7th International Workshop on Managing Technical Debt (MTD), Bremen, 2015, pp. 53-56, doi: 10.1109/MTD.2015.7332625.

[6] R. J. Eisenberg, "A threshold-based approach to technical debt", ACM SIGSOFT Software Engineering Notes, 37 (2), pp. 1 - 6, ACM, 2012.

[7] S. Charalampidou, A. Ampatzoglou, A. Chatzigeorgiou, A. Gkortzis, and P. Avgeriou, "Identifying Extract Method Refactoring Opportunities Based on Functional Relevance", IEEE Transactions on Software Engineering, 43 (10), pp. 954-974, 2017.

[8] M. Fokaefs, N. Tsantalis, A. Chatzigeorgiou, and J. Sander, "Decomposing object-oriented class modules using an agglomerative clustering technique", In 2009 IEEE International Conference on Software Maintenance, pp. 93-101, IEEE, 2009.

[9] W. Cunningham, "The WyCash Portfolio Management System," in Addendum to the proceedings on Object-oriented programming systems, languages, and applications, pp. 29-30, 1992.

[10] P. Kruchten, R. L. Nord and I. Ozkaya, "Technical Debt: From Metaphor to Theory and Practice," in IEEE Software, vol. 29, no. 6, pp. 18-21, Nov.-Dec. 2012, doi: 10.1109/MS.2012.167.

[11] J. Yli-Huumo, A. Maglyas, and K. Smolander, "How do software development teams manage technical debt?–An empirical study", Journal of Systems and Software, 120, pp. 195-218, 2016.

[12] J. Conejero et al., "Early evaluation of technical debt impact on maintainability", Journal of Systems and Software, vol. 142, pp. 92-114, 2018.

[13] C. Seaman and Y. Guo, "Measuring and monitoring technical debt", Advances in Computers, Elsevier, 82, pp. 25 - 46, 2011.

[14] Ar. Ampatzoglou, A. Ampatzoglou, P. Avgeriou, and A. Chatzigeorgiou, "A Financial Approach for Managing Interest in Technical Debt", International Symposium on Business Modeling and Software Design (BMSD'15), Milan, Italy, 6 – 8 July 2015.

[15] Ar. Ampatzoglou, N. Mittas, A.A. Tsintzira, A. Ampatzoglou, E.M. Arvanitou, A. Chatzigeorgiou, P. Avgeriou, and L. Angelis, "Exploring the Relation between Technical Debt Principal and Interest: An Empirical Approach," *Inf. Softw. Technol.*, vol. 128, p. 106391, Dec. 2020,

[16] D. Heaton and J. Carver, "Claims about the use of software engineering practices in science: A systematic literature review", Information and Software Technology, vol. 67, pp. 207-219, 2015.
[17] R. Farhoodi, v. Garousi, d. Pfahl and j. Sillito, "development of scientific software: a systematic mapping, a bibliometrics study, and a paper repository", International Journal of Software Engineering and Knowledge Engineering, vol. 23, no. 04, pp. 463-506, 2013.
[18] M. T. Sletholt, J. Hannay, D. Pfahl, H. C. Benestad, and H. P. Langtangen, "A literature review of agile practices and their effects in scientific software development," in Proceeding of the 4th international workshop on Software engineering for computational science and engineering - SECSE '11, 2011.
[19] M. Fowler, K. Beck, J. Brant, W. Opdyk, and D. Roberts, "Refactoring: improving the design of existing code", ser. In Addison Wesley object technology series, Addison-Wesley, 1999.
[20] T. McCabe, "A Complexity Measure", Transactions on Software Engineering, 2 (4), pp. 308-320, IEEE, 1976.
[21] E. M. Arvanitou, A. Ampatzoglou, A. Chatzigeorgiou, and P. Avgeriou, "A method for assessing class change proneness", In Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering, pp. 186-195, 2017.
[22] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design", Transactions on Software Engineering, , 20 (6), pp. 476-493, IEEE, 1994.
[23] R.C. Martin, "Agile software development: principles, patterns and practices", Prentice Hall, New Jersey, 2003.
[24] M. Riaz, E. Mendes, and E. Tempero, "A systematic review of software maintainability prediction and metrics," 3rd International Symposium on Empirical Software Engineering and Measurement, pp. 367-377, IEEE, 2009.
[25] R. Marinescu, "Detection strategies: metrics-based rules for detecting design flaws", In Proceedings of the 20th International Conference on Software Maintenance, pp. 350-359, IEEE, 2004.
[26] K. A. Ferreira, M. A. Bigonha, R. S. Bigonha, L. F. Mendes, and H. C. Almeida, "Identifying thresholds for object-oriented software metrics", Journal of Systems and Software, 85(2), pp. 244-257, 2012.
[27] R. Marinescu, "Detection strategies: metrics-based rules for detecting design flaws," in 20th IEEE International Conference on Software Maintenance, 2004. Proceedings., Sep. 2004, pp. 350–359, doi: 10.1109/ICSM.2004.1357820.
[28] W. Li and S. Henry, "Object-oriented metrics that predict maintainability", Journal of Systems and Software, 23 (2), pp. 111-122, Elsevier, 1993.
[29] A. Mori, G. Vale, M. Viggiato, J. Oliveira, E. Figueiredo, E. Cirilo, P. Jamshidi, C. Kastner, "Evaluating domain-specific metric thresholds: an empirical study," in Proceedings of the 2018 International Conference on Technical Debt, New York, NY, USA, May 2018, pp. 41–50.
[30] G. Vale, E. Fernandes, and E. Figueiredo, "On the proposal and evaluation of a benchmark-based threshold derivation method," Softw. Qual. J., vol. 27, no. 1, pp. 275–306, Mar. 2019.
[31] D. Cedrim et al., "Understanding the impact of refactoring on smells: a longitudinal study of 23 software projects," in Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, New York, NY, USA, Aug. 2017, pp. 465–475.
[32] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," in Proceedings of the 31st International Conference on Software Engineering, New York, NY, USA, May 2009, pp. 287–297.
[33] T. De Marco, "Structured Analysis and System Specification", Yourdon Press Computing Series, 1979.