Practitioners' Perspective on Practices for Preventing Technical Debt Accumulation in Scientific Software Development

Elvira-Maria Arvanitou[1], Nikolaos Nikolaidis[1], Apostolos Ampatzoglou[1], Alexander Chatzigeorgiou[1]
[1]*Department of Applied Informatics, University of Macedonia, Greece*
e.arvanitou@uom.edu.gr, it14189@uom.edu.gr, a.ampatzoglou@uom.edu.gr, achat@uom.edu.gr

Keywords: software engineering practice; technical debt; scientific software development; prevention.

Abstract: Scientific software development refers to a specific branch of software engineering that targets the development of scientific applications. Such applications are usually developed by non-expert software engineers (e.g., natural scientists, biologists, etc.) and pertain to special challenges. One such challenge (stemming from the lack of proper software engineering background) is the low structural quality of the end software—also known as Technical Debt—leading to long debugging and maintenance cycles. To contribute towards understanding the software engineering practices that are used in scientific software development, and investigating whether their application can lead to preventing structural quality decay (also known as Technical Debt prevention); in this study, we seek insights from professional scientific software developers, through a questionnaire-based empirical setup. The results of our work suggest that several practices (e.g., *Reuse* and *Proper Testing*) can prevent the introduction of Technical Debt in software development projects. On the other hand, other practices seem as either improper for TD prevention (e.g., *Parallel / Distributed Programming*), whereas others as non-applicable to the branch of scientific software development (e.g., *Refactorings* or *Use of IDEs*). The results of this study prove useful for the training plan of scientists before joining development teams, as well as for senior scientists that act as project managers in such projects.

1 INTRODUCTION

Scientific software development refers to the end-to-end (from requirements analysis to deployment and maintenance) construction lifecycle of software applications used for scientific purposes (e.g., physics, biology, medical analysis, and data science). The necessity for developing scientific software has emerged due to the need for continuous experimentation and validation of research outcomes (e.g., simulations, or cases studies) before the publication of results (Birdsall and Langdon, 1991). Nevertheless, such a continuous experimentation leads to continuous maintenance (i.e., small incremental developments, debugging, and bug fixing cycles); which, by considering the usually long execution time of such software (in the common case executed upon big data), can lead to long delays in the scientific process, if maintenance is not successful.

During the last decade, in "*traditional*" software engineering, the term Technical Debt (TD) (Cunningham, 1992) has emerged so as to capture the efficiency of maintenance process, both in terms of improving the maintainability of the software, as well as, of costs occurring due to low maintainability of software (Avgeriou et. al., 2016). In every system, the accumulation of TD is inevitable, since the development of zero-TD systems is not financially viable; and therefore: non-realistic (Eisenberg, 2012). Consequently, the TD that is accumulated in a software system needs to somehow be controlled, so as to reduce its negative impacts. In the literature, two TD reduction methods have been introduced: *TD repayment* (e.g., through refactoring) (Li et al., 2015) and *TD prevention* (e.g., through writing clean new code) (Digkas et al., 2022). By contrasting the two options, TD prevention seems more fitting for the domain of scientific software development, since: (a) dedicated refactoring sessions are not usual in this context; (b) there is limited refactoring support for the most common programming languages in the field (usually non-OO languages); and (c) scientific software is usually developed by scientists, without a strong background in software development—refactoring might be a non-trivial task for them.

Based on the above, in this paper we aim to empirically understand and discuss how TD accumula-

tion could be prevented in the field of scientific software development. To achieve this goal, we: (a) seek for Software Engineering (SE) practices that are used while developing scientific software; (b) identify the most common causes of introducing TD; and (c) search for a mapping between the two. For identifying SE practices relevant to the scientific software development community, we refer to a very recent secondary study in the field conducted by Arvanitou et al. (2021); whereas for spotting potential causes of TD accumulation, we refer to the outcomes of the InsighTD project[1] (Rios et al., 2020). Given the above, our main contribution is "*the exploration of which SE practices applied in scientific software development can be used for preventing TD accumulation".*

To achieve this goal, we have performed a questionnaire-based study on 5 scientific software development organizations, aiming at understanding: (a) which SE practices the developers are familiar with; (b) which SE practices they use more often; and (c) which causes of TD accumulation can be hindered by applying each practice. Obtaining the aforementioned knowledge can advance scientific software development practice, since: (a) it can guide the necessary SE-related training of scientists before joining development teams; (b) it can help senior scientists playing the role of the project managers on which SE practices they need to impose in their development teams; and (c) it can contribute towards the development of an SE culture and a TD awareness in the community, by noting the causes of TD accumulations and how they can be prevented.

The rest of the paper is organized as follows: in Section 2 we provide all necessary background information to understand the main concepts of this study: (a) the SE practices used in scientific software development; and (b) the root causes of TD accumulation. Next, in Section 3 we describe the setup of the conducted empirical study; whose results we present in Section 4. In Section 5, we provide a discussion based of the results, aiming to deliver the main contribution of this work: i.e., how TD accumulation can be prevented in the scientific software development community. In Section 6, we discuss tentative threats to validity, and in Section 7 we conclude the paper.

2 BACKGROUND CONCEPTS

In this section, we present the necessary required background information to establish a better understanding of this paper. To this end, in Section 2.1 we present the SE practices that we have explored, whereas in Section 2.2 the causes that can lead to TD accumulation that can be considered for prevention purposes.

2.1. Software Engineering Practices for Scientific Software Development

In this section, we present the most common SE practices for scientific software development, based on the literature. More specifically, Arvanitou et al. (2021) performed a mapping study to investigate the current state-of-research and –practice on the use of SE practices in scientific software development. Table 1 presents the top-25 most used SE practices applied in this domain and the definition of each practice (Arvanitou et al., 2021).

Table 1: Top-25 Software Practices

Practice	Definition
Reuse of Library	Use pieces of software (packaged in the form of a library) in software systems, other than the one that they have been originally developed for. This practice is expected to lower testing effort and improve development productivity.
Use of API	Use of a set of predetermined available functionalities, available through a protocol that allows two applications or services to communicate with each other. Similarly to before, this practice can reduce bugs and increase productivity.
MDE	Focuses on the construction of a software model (e.g., a diagram that specifies how the software system should work) which is automatically transformed to code. MDE improves the ability of novice developers to produce code by handling more high-level artifacts.
Skeleton Programming	Use of predefined generic program building blocks for frequently occurring computation patterns (e.g., data-parallel map, reduce, scan, stencil) for which efficient platform-specific implementations exist. Expected to improve performance of the application.
AOP	A programming paradigm that aims to increase modularity by allowing the separation of cross-cutting concerns.
OOP	A programming paradigm based on the concept of objects, containing data and methods. OOP introduces various benefits, the most prominent one being the organization of code to entities and actions, similarly to the human perception.

[1] http://www.td-survey.com/

Practice	Definition
Task Scheduling	The ability to schedule when a task will start and stop. The approach is used in software development, mostly in terms of increasing fault tolerance.
Parallel Programming	A programming paradigm that enables the simultaneous execution of multiple instructions to speed-up the solution of a computational problem.
Process Improvement	Any method that can be used for making the software development process more efficient, e.g., speed-up the development process, producing software with less faults, etc.
CBSE	An approach that focuses on the design and development of computer-based systems with the use of reusable software components. These components are usually acquired off-the-shelf and are reused through their public API. This practice is expected to lower testing effort and improve development productivity.
Development Framework	A software providing generic functionality, which is accompanied by additional user-written code, can provide application-specific software. Frameworks may include libraries, compilers, toolsets, and APIs. Similarly to before, the use of frameworks reduces faults and speeds-up development.
Testing	Approaches that check if the actual implementation of the software matches the expected requirements. The outcome of such approaches is the identification of defects, before the software becomes operational.
Requirements, Design, Architecture	The process through which the outcomes of various development phases are specified in documents. The usual artifacts produced are models (diagrams or texts) that describe how the system is expected to be developed. Such models enable the understanding of code structure, reducing the time required to understand code while maintaining the system.
Quality Assurance	A procedure that aims at ensuring the quality of software products or services provided to the customers by an organization. When quality assurance focuses on maintainability, the TD preventive power is obvious.
Quality Metric	An approach that focuses on the quantification of quality aspects of the product, process, and project (e.g., size, complexity, test coverage). The use of metrics can help in: (a) monitor the trend of software quality; (b) compare tentative solutions for reuse; and (c) identify spots in the code that might be suboptimal in terms of quality.

Practice	Definition
TDD	A software development approach in which test cases are developed before coding, so as to specify and validate what the code will do. TDD ensures that are test are passed before the operation of the software, reducing the chance of facing bugs while executing the software.
CI / CD	A development approach in which the engineer automates the integration of code changes from multiple contributors into a single software project. Such processes support automated testing, building, and deployment of the solutions.
Pair Programing	A software development approach in which two programmers work together at one workstation (one screen, keyboard and mouse among the pair) so as to increase early fault identification and reduce development delays.
Refactoring	A technique for changing an existing code, altering its internal structure without changing its external behavior. Refactoring is the most common way for repaying TD.
Pattern	Well-known and established in practice solutions to a commonly occurring problem in software design or coding. Also applicable to architecture, as well as for guaranteeing specific qualities: security, or low memory patterns.
Advanced IDE	A software suite that enables programmers to consolidate the different aspects of writing a computer program. Advanced IDEs provide various kinds of assistance, such as autocomplete, refactoring, automatic styling, etc.
DSL	A programming language with a higher level of abstraction optimized for a specific class of problems. Such languages are closer to the domain and boost productivity; nevertheless, their generic use is limited.
Code Generation	The automated synthesis of software assets, such as documentation or models, to produce code. Code generation can speed-up development and enable more novice developers to produce code.
Collab. Software Development	An application that helps people working on a common task to succeed in their goals. The most known example of this category is collaborative source code development (e.g., with Git). Speeds-up development and eases management.
Specific Programing Language	A set of commands, instructions, and other syntax is used to create a software program. Some languages (e.g., C, C++, Java, R, and Python) are proven as more fitting for scientific software development.

2.2 TD Prevention Causes

In this section, we present the most common causes of TD based on the literature. As explained by Rios et al. (2020), this list can help development teams to identify actions that could prevent the introduction of TD items in the first place. Thus, it is worthwhile to understand the causes that could lead a development team to accumulate TD, and propose mitigation actions as prevention measures. The four studies that are dealing with identifying possible causes of TD accumulation are outlined below.

Martini et al. (2014) performed a multiple-case embedded study in seven sites at five large organizations to investigate the current causes for the accumulation of architectural TD (ATD). As a result of this study, the authors provided a taxonomy of causes and their influence in the accumulation of ATD. In addition, Martini and Bosch (2017) conducted a case study in order to investigate (a) the most dangerous ATD items in terms of effort paid later; (b) the effects triggered by such ATD items; and (c) if there are sociotechnical patterns of events that trigger the creation of ATD. The results suggested that TD items can be contagious, causing other parts of the system to be contaminated with the same problem, which may lead to nonlinear growth of interest. The authors also presented a model of ATD effects that can be used for TD repayment prioritization.

Yli-Huumo et al. (2016) performed a case study to investigate the role of technical debt management in software development. In particular, the goal of this study was to explore the causes of TD accumulation, as well as its effects, and the strategies that are being used for technical debt management. The results of this study suggested that the reasons for incurring TD were management decisions that were made in order to reach deadlines, or unknowingly due to lack of knowledge.

Finally, as a more recent work in this area, Rios et al. (2020) conducted an industrial survey in different countries in order to investigate the trends in the TD area including the causes and the effects of TD. 107 practitioners from 11 countries joined in the survey. The results of this study suggested that most of the practitioners were familiar with the concept of TD. As a final outcome Rios et al. (2020) identified 78 causes that lead to TD occurrence. Out of them, we focus on the most cited causes that lead to the accumulation of TD. According to Rios et al. (2020), the most cited causes of TD are presented below:

- ***Deadline***—A certain period of time defined by the team, project manager and / or customer to deliver a determined activity, feature or product. *Example*: "The rush of managers (customers) that want to receive something working asap";
- ***Inappropriate Planning***— Refers to problems in the planning of software development activities, treated as a project. *Example*: "Lack of prioritization of activities";
- ***Lack of Knowledge***—Refers to lack of knowledge about specifications of the project, the unfamiliarity with any activity or artifact of the project, and the usage, the operation or the purpose of a particular technology. *Example*: "Lack of testing knowledge in team";
- ***Lack of Defined Development Process***— Refers to the lack of a sustainable methodology aimed at creating and maintaining guides that would increase the productivity of the software development software team. *Example*: "Lack of a followed processes";
- ***Lack of Tests***—Means that the project has not been tested at all, or that the testing is not sufficient—not covering all requirements. *Example*: "Lack of (functional) testing"
- ***Ineffective Project Management***— Refers to inadequate management during the complete software development lifecycle. *Example*: "Not following planning"
- ***Lack of Qualified Professionals***— Occurs when unprepared professionals perform a certain activity or because of lack of professionals prepared to carry it out. *Example*: "Absence of specialist to carry out specific activities";
- ***Lack of Experience***— Refers to the lack of experience, obtained through the practice in certain software development activities. *Example*: "Lack of experience of programmers";
- ***Outdated or Incomplete Documentation***—Occurs when software documentation is outdated, unfinished, or simply missing in the software project. *Example*: "Incomplete documentation";
- ***Lack of Commitment***— Non-professional commitment of stakeholders (usually software engineers) to fulfil the tasks assigned to them along the whole software development lifecycle. *Example*: "Stakeholders not engaged";
- ***Poor Design***— Refers to poorly designed project, suffering from example from high coupling. *Example*: "Poorly designed database structure".

3 CASE STUDY DESIGN

The case study reported in this section was executed as part of the EXA2PRO project[2]. EXA2PRO was an EU-funded FET project aiming (among other) to explore the potential of applying TDM approaches in High Performance Computing, and cultivate a structural quality culture in corresponding development teams. The study was designed and is reported, based on the guidelines by Runeson et al. (2012).

3.1 Research Objectives / Questions

The goal of this study expressed in terms of the Goal-Question-Metric (GQM) approach (Basili, 1992) is formulated as follows: "***analyze*** *the software engineering (SE) practices* ***for the purpose of*** *characterization* ***with respect to*** *their ability to prevent the accumulation of technical debt; as well as their usefulness (acquaintance of developers with them and applicability in scientific software development)* ***from the point of view of*** *scientists that develop software*". Based on this, we have derived three RQs:

RQ1: How familiar are the scientists that develop scientific software to SE practices?
RQ2: How often do practitioners use SE practices?
RQ3: Can the use of SE practices prevent the accumulation of TD?

The answer of the ***RQ1*** aims to identify the current knowledge of scientists that develop scientific software on SE practices. The answer to this research question will shed light on the usefulness of the SE practices, in the sense that practices to which the scientists are not familiar with, are having less chances of being applied in practice. The answer of the ***RQ2*** explores the frequency of the use of SE practices. The answer to this research question is complementary to RQ_1, in the sense that it unveils if a specific practice is applicable to the pilot cases, and in scientific software development in general. Finally, the ***RQ3*** aims to explore the link between SE practices and the causes of TD. More specifically, we explore if the use of SE practice could prevent the causes of TD, and consequently if their application can prevent the accumulation of TD in software systems.

3.2 Participants Selection

To answer the aforementioned questions, we performed a questionnaire-based study in cooperation with the following organizations, related to the development of scientific software—see Table 2.

Table 2: Participants Selection

Organization	Examples of Scientific Software
JUELICH	Lattice Quantum Chromodynamics (LQCD) refers to a class of applications which concerns itself with simulation of the theory of strong interactions KKRnano is a massively parallel code performing Density Functional Theory (DFT) simulations.
CERTH	CO_2Capture, which is used for inventing new materials for CO2Capture and providing economically viable installation solutions for industries
CNRS	Metalwalls which is a supercapacitor simulation from the energy storage application domain
INRIA	INRIA is a research organization that focuses on scientific software development, by providing tools that enable task scheduling through, e.g., StarPU
LIU	LIU is a university that focuses on research activities concerning the development of high-level software abstractions and composition frameworks for scientific software. The most notable product of LIU is SkePU, which is well-established in the scientific software development domain.

For each organization, we asked our contact point to forward the email to at least 10 scientific software developers per organization. To comply with GDPR, we informed the participants of the survey that: (a) the results of the study will be made available to them in an aggregate form in case they are interested; (b) will only be published in an aggregated form; (c) each participant should complete the questionnaire only if he / she provides his/her consent, and (d) their data will be erased upon participants requests.

3.3 Data Collection and Analysis

The data collection method was an unsupervised *questionnaire-based*, aiming to provide input to all research questions. Despite the questionnaire-based nature of this work, we cannot characterize this work as a survey, since we have reached a limited population. The questionnaire was organized into three parts, one per RQ: the structure of the questionnaire is presented in Figure 1 and is repeated for all *SE practices* presented in Table 1. The questionnaire was provided online, in the form of Google Forms[3].

[2] http://www.exa2pro.eu

[3] https://forms.gle/KCemCmc8Gcnu54Us5

Is "SE practice" an understandable SE practice?
(Very Difficult to Understand → Very Easy to Understand) – 5 scales

How often do you use "SE practice"?
(Very Scarcely → Very Often) – 5 scales

Can "SE practice" prevent the accumulation of TD, caused by...
- short deadlines?
- inappropriate planning?
- lack of knowledge?
- lack of a defined development process?
- the lack of tests?
- ineffective project management?
- lack of qualified professionals?
- lack of experience?
- outdated or incomplete documentation?
- lack of commitment?
- poor design (e.g., lack of refactorings, etc.)?

Other Comments (free text)

Figure 1: Structure of Questionnaire

The use of Google Forms provided us the opportunity to easily setup the survey instrument, whereas all responses were managed automatically. The main benefit of this strategy is that no errors during the recording of the responses can be introduced. At the end of the data collection process, the dataset consists of 75 columns (3 questions x 25 SE Practices) and 31 rows (responses). To answer the research questions, we performed a quantitative assessment based on the questionnaire data. The dataset has been analyzed using descriptive statistics and graphs.

4 RESULTS

In this section, we present the results of this study based on the research questions that have been described in Section 3.1. We note that in this section we only present the raw results of our investigation, which are cumulatively discussed in Section 5.

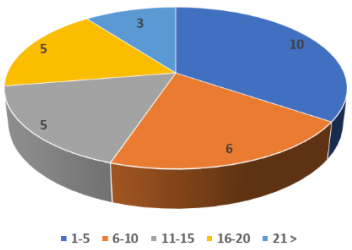

Figure 2: Study Demographics

Figure 2 depicts the experience of the participants, which is measured in years. More specifically, we can observe that approximately 50% of the participants have more than 10 years' experience as scientific software developers.

Familiarity & Usage of SE Practices by Scientific Software Developers (RQ_1 / RQ_2)

To investigate the knowledge of scientists and the usage frequency of SE practices, we asked them how familiar they are with the top-25 SE practices, and how often they use these practices. Since the extracted information is vast, to present it comprehensively, we have preferred to discuss the extreme cases only. More specifically, we present:

- ***Favourable SE practices*** with which developers are highly familiar and use them in practice. In this category we have classified: "Reuse of Software Libraries", "Application Programming Interfaces", "Object-Oriented Programming", "Task-Based Programming", "Continuous Integration", "Collaborative Software Development, and "Parallel / Distributed Programming".
- ***Less applicable SE practices*** that developers know, but they prefer not to use them. In this category we have classified: "Testing", "Refactoring", "Integrated Development Environment", and "Code Generation"
- ***Less familiar SE practices*** only few developers are aware of. In this category we have classified: "Model Driven Engineering", "Aspect Oriented Programming", "Paired Programming", and "Process Improvement Methods".

In Figures 3-6 we visualize the results for RQ_1 and RQ_2, in the form of grouped bar charts for two favorable, one less applicable, and one less familiar practice. The bar charts correspond to the frequency of each Likert scale value, in terms of familiarity to the practice (blue bars); and the usage frequency (orange bar).

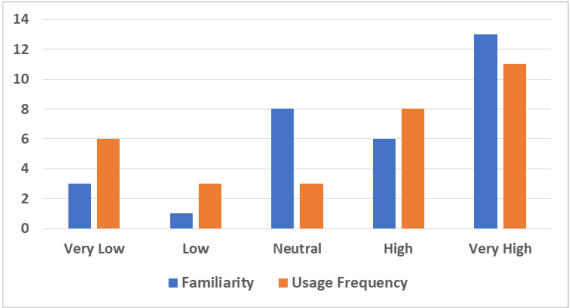

Figure 3: Reuse of Software Libraries

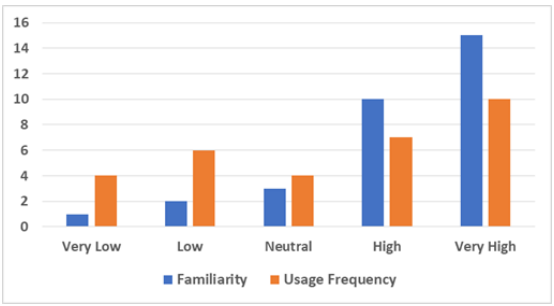

Figure 4: Application Programming Interfaces

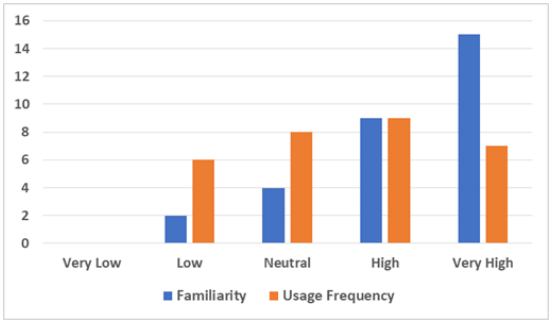

Figure 5: Testing

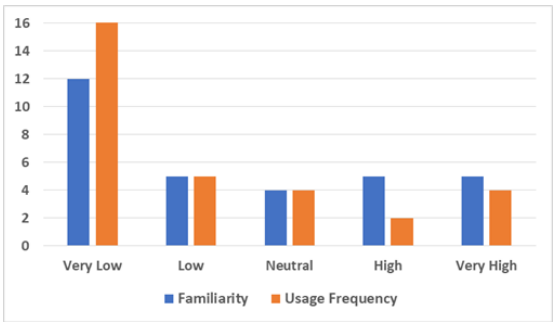

Figure 6: Model Driven Engineering

TD Prevention through SE Practices (RQ_3)

To answer RQ_3, we have visualized the collected data through a bubble chart (see Figure 7). We note that for answering RQ_3, we focused on *Favorable SE practices* and *Less familiar SE practices*. The bubble chart can be read in two ways: horizontally or vertically. The horizontal analysis highlights the SE practices that mitigate most of TD causes, whereas the vertical analysis denotes the SE practices that can be used for mitigating specific causes.

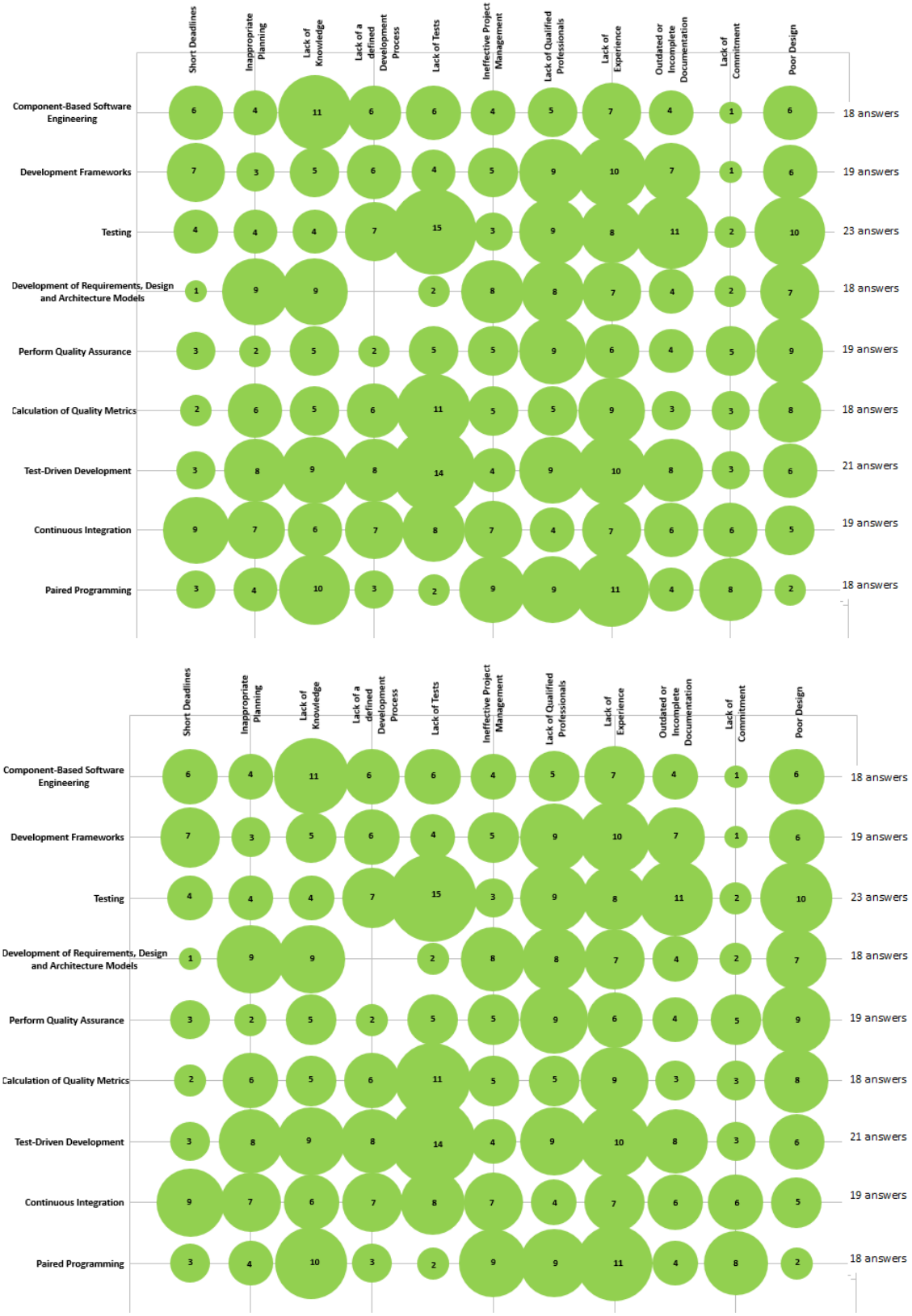

Figure 7: Mapping of SE Practices and Causes of TD accumulation

5 DISCUSSION

Which Causes can each SE Practice Mitigate? As a threshold to answer this question, we have set the 10 answers; i.e., that at least 10 practitioners agree that a specific practice can mitigate some TD cause. Below we list the SE practices that can mitigate at least 2 causes of TD. As the most prominent SE practices (that resolve 2 causes of TD), we identify two: The application of "*Reuse of Software Library*" can mitigate in total 3 causes of TD: "*short deadlines*", "*lack of knowledge*" or "*lack of experience*"; suggesting that reuse is highly relevant for teams with low programming experience (which is the usual case for scientific software developers). The application of "*Testing*" practice can also mitigate 3 causes of TD: "*lack of tests*", "*outdated or incomplete documentation*" or "*poor design*".

The following SE practices can resolve two causes of TD. In particular, the application of "*Use of Specific Programming Language*" practice can mitigate the TD caused by "*lack of knowledge*", "*lack of qualified professionals*" or "*lack of experience*". The application of "*Skeleton Programming*" practice can mitigate the TD caused by "*lack of knowledge*" or "*lack of experience*". The application of "*Process Improvement Methodologies*" practice can mitigate the TD caused by "*short deadlines*" or "*inappropriate planning*". The application of "*Test-Driven Development*" practice can mitigate the TD caused by "*lack of tests*" or "*lack of experience*". The application of "*Paired Programming*" practice can mitigate the TD caused by "*lack of knowledge*" or "l*ack of experience*". Finally, the application of "*Design/Code Patterns*" practice can mitigate the TD caused by "*lack of experience*" or "*poor design*".

Based on the above, we encourage scientific software developers to apply Reuse of Software Libraries and Process Improvement Methodologies, as well as to work in a Paired-Programming manner. The application of these practices is expected to mitigate in total 40% of TD causes.

Which Causes of TD Cannot be Mitigated with the Identified SE practices? To answer this question, we focus on the TD causes that are reported to be mitigated by only one or no SE practice. "*Inappropriate Planning*", "*Ineffective Project Management*", and "*Lack of Qualified Professionals*" are mitigated by only one of the investigated SE practices. Additionally, "*Lack of Defined Development Process*" and "*Lack of Commitment*" are not reported to be mitigated by any of the investigated SE practices.

Thus, more research and practical focus is required in the project management-related causes of TD, since their mitigation seems neglected, compared to more technical causes.

Other Findings. According to Rios et al. (2020), the most common causes of TD are "*lack of experience*" and "*lack of knowledge*". The SE practices that resolve these causes of TD accumulation are: "*Reuse of Software Libraries*", "*Skeleton Programming*", and "*Paired Programming*". Therefore, we need to highlight the importance of these practices as well. Another interesting observation compared to the results from RQ_1 and RQ_2 with RQ_3 is related to the SE practice "*Parallel / Distributed Programming*". Although the results suggest that the scientists are familiar with this practice and use it very often, they use it for other reasons and not for mitigating TD. More specifically, one researcher wrote that "*This practice is something we have to use to make our program parallel, but it does not help in SE*", whereas another researcher wrote that "*it makes everything else more difficult. It only helps with getting better performance*".

6 THREATS TO VALIDITY

While designing this study, we have identified several threats to validity. First, regarding conclusion validity, all interpretations are tentative ones, since (by definition) surveys cannot support causality, but only report trends and general beliefs in the state-of-practice. Additionally, the sample of this study is a bit narrow compared to other questionnaire-based studies; however, it could not be expanded to the complete software engineers' population, since we focus on scientific software development. Nevertheless, we note that the wide-spread of the sample to many organizations, that vary across EU countries guarantee to some extent the generalizability of the results.

Furthermore, we acknowledge that repeating the study with a different set of scientists might yield different results; however, the study design is completely replicable since all data collection instruments and procedures are presented transparently in Section 3. Finally, a threat to construct validity stems from the fact that we presented to the participants only elements retrieved from the literature or existing tools; therefore, we might have missed other aspects (e.g., other SE practices or TD accumulation causes) that they consider important, but were not listed in tentative answers, considering also the lack of open-ended questions.

7 CONCLUSIONS

In this study, we focused on the scientific software development domain, which is a sub-field of software engineering, limited to the implementation of software for research purposes. The goal of the study was to identify which software engineering practices can be used in scientific software development to prevent the accumulation of technical debt. On the one hand, the study of SE practices in this domain is important, since usually scientific developers are not software engineers; on the other hand, TD management is also considered as highly relevant for the domain, since maintenance of such applications is frequent, whereas also possibly miss-execution (due to errors) is very costly.

To achieve this goal, we have performed a questionnaire-based study with approximately 30 scientific software developers, from 5 organizations spread across Europe. The results of the study unveiled that several SE practices, such as Reuse or Proper Testing, can prevent the accumulation of TD. On the other hand, other practices seem as either irrelevant to TD prevention (e.g., Parallel Programming), or as non-applicable to scientific software development (e.g., Refactorings). These findings can be quite useful in practice, since the most fitting practices can: (a) be promoted in the training plan of scientists; (b) be encouraged to be used in practice by technical managers. Finally, we believe that even the process of executing such studies contributes towards the development of an SE culture in scientific software development, pushing the community to move towards more systematic engineering processes.

ACKNOWLEDGEMENTS

This work has received funding from two European Union's H2020 research and innovation programmes, under grant agreements: 871177 (SmartCLIDE) and 801015 (EXA2PRO). The work of Dr. Arvanitou was financially supported by the action "Strengthening Human Resources Research Potential via Doctorate Research" of the Operational Program "Human Resources Development Program, Education and Lifelong Learning, 2014-2020", implemented from State Scholarship Foundation (IKY) and co-financed by the European Social Fund and the Greek public (National Strategic Reference Framework (NSRF) 2014–2020). The work of Mr. Nikolaidis is funded by the University of Macedonia Research Committee as part of the "Principal Research 2020" funding program.

REFERENCES

Arvanitou, E. M., Ampatzoglou, A., Chatzigeorgiou, A. and Carver, J. C. (2021). Software engineering practices for scientific software development: A systematic mapping study, *Journal of Systems and Software*. 172.

Avgeriou, P., Kruchten, P., Ozkaya, I. and Seaman, C. (2016). Managing Technical Debt. in Software Engineering (Dagstuhl Seminar 16162). Dagstuhl Reports. 6 (4). pp. 110–138.

Basili, V. R. (1992). Software modeling and measurement: the Goal/Question/Metric paradigm.

Birdsall, C. K. and Langdon, A. B. (1991). Plasma Physics via Computer Simulation. *the Adam Hilger Series on Plasma Physics*. Adam Hilger, New York.

Cunningham, W. (1992). The WyCash Portfolio Management System. *7th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '92)*. Vancouver, British Columbia, Canada, October 18-22.

Digkas, G., Chatzigeorgiou, A., Ampatzoglou, A. and Avgeriou, P. (2022). Can Clean New Code Reduce Technical Debt Density? *Transactions on Software Engineering*, IEEE Computer Society.

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

Li, Z., Avgeriou, P. and Liang, P. (2015). A systematic mapping study on technical debt and its management. *Journal of Systems and Software*. Elsevier. 101, pp. 193-220.

Martini, A. and Bosch, J. (2017). On the interest of architectural technical debt: Uncovering the contagious debt phenomenon. *Journal of Software: Evolution and Process*, 29(10).

Martini, A., Bosch, J. and Chaudron, M. (2014). Architecture technical debt: Understanding causes and a qualitative model. *40th EUROMICRO Conference on Software Engineering and Advanced Applications*. pp. 85-92.

Rios, N., Spínola, R. O., Mendonça, M. and Seaman, C. (2020). The practitioners' point of view on the concept of technical debt and its causes and consequences: a design for a global family of industrial surveys and its first results from Brazil. *Empirical Software Engineering*. pp. 1-72.

Runeson P, Host M, Rainer A, Regnell B. Case study research in software engineering: Guidelines and examples. Hoboken: Wiley; 2012.

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