Investigating the Relationship between Co-occurring Technical Debt in Python

Jie Tan, Daniel Feitosa, Paris Avgeriou
Bernoulli Institute for Mathematics, Computer Science and Artificial Intelligence
University of Groningen
Groningen, The Netherlands
{j.tan, d.feitosa, p.avgeriou}@rug.nl

***Abstract*—Technical debt (TD) reflects issues that may negatively affect software maintenance and evolution. There is currently little evidence on how the different types of TD co-occur; for example, how code smells and design smells affect the same part of the system. This paper investigates how different types of TD co-occur, as well as the time period of the co-occurrence. To that end, we analyzed the co-occurring associations between five types of TD, captured in 42 SonarQube rules, in 3862 files of 20 Python projects from the Apache Software Foundation. We found that this phenomenon is dominant, affecting more than 90% of Python files. We also found that Documentation Debt and Test Debt appear in the majority of the files, although it seems to be mostly by coincidence. Finally, we noticed that co-occurrence of TD seems to happen very quickly: co-occurring issues tend to be introduced within the same week. But once it does happen, it is hard to get rid of. These results can benefit both researchers and practitioners by: aiding the prioritization of TD remediation; leading to novel tools for detecting co-occurring TD and warning potential issues; shedding further light on the explanation of how TD is introduced and can be mitigated.**

***Index Terms*—technical debt; co-occurrence; Python; Apache Software Foundation**

I. Introduction

Technical debt (TD) reflects shortcomings in a software system that can potentially harm its maintainability and evolvability [1]. Different types of TD are related to different artifacts of the software development life cycle. The majority of work on TD is related to the source code, but there is also work associated with other types of debt, related to documentation, design, defects and testing [2], [3].

The vast majority of research on TD has focused on investigating the effect of *individual* instances of TD, e.g., code smells [4]. More recently, researchers have been looking into the interactions among different TD instances, when they *co-occur* in the same component, class, or file. For example, one study found six pairs of co-occurring code smells that frequently appear together in open-source software systems and elaborated on the extent of this phenomenon as well as how such co-occurrences are introduced and removed by developers [5]. Other studies confirmed that co-occurring TD poses a great risk as it can intensify the negative consequences caused by individual issues [6], [7].

Motivation. Although the related work has investigated the negative impact of co-occurring TD instances on maintenance, they have two main limitations.

First, there is very little work beyond code smells, on how other types of TD co-occur (e.g., Design Debt or Documentation Debt). A broader understanding of the different types of debt in a project is essential to better inform the involved stakeholders on various maintenance liabilities, especially because the different types of debt are related. For example, one may decide to document the code at a later moment (i.e., Documentation Debt) and this decision may lead to further creation of unnecessarily complex components (i.e., Design Debt) due to poor insight into their implementation.

Second, current research does not deal with the time interval between a co-occurrence appearing and disappearing. This may hint at possible reasons for TD co-occurrences and provide information for prioritizing their remediation. For example, if one type of debt is always followed by the appearance of a second type very soon after the first, the first may be causing the second. That could, at the very least, result in monitoring the first type from an early stage. Conversely, co-occurring issues that seem to be removed almost at the same time, can be targeted together for more effective maintenance.

Aim and scope. This paper reports on an empirical study that investigates the relationship between different, co-occurring issues and types of technical debt. The study focuses on the issues and types of TD that co-occur frequently and on the interval between their introduction and removal. We consider approx. 28,200 commits from 20 Python projects from the Apache Software Foundation (ASF) and investigate five types of TD: Design, Code, Defect, Documentation and Test Debt.

To detect TD, we use SonarQube, an open-source tool that measures quality and technical debt. We selected this tool for two main reasons: (1) it is being widely used in industry[1] as well as in the literature of TD [8]; (2) it can track the evolution of technical debt by analyzing multiple versions of projects. SonarQube detects TD by identifying violations of a number of rules; different rules correspond to different types of TD. Limitations of using SonarQube are reported in Section V.

This work was supported by ITEA3 and RVO under grant agreement No. 17038 VISDOM (https://visdom-project.github.io/website/).

[1]https://www.sonarsource.com/customers/

We have selected projects in Python, as it has grown substantially over the years and is currently ranked as the 3rd most popular programming language[2]. Although developers can benefit from the flexibility and conciseness brought by the dynamic features of Python, they have to spend extra effort on software maintenance and software quality improvement [9]. Thus, our results can guide developers specifically in managing TD for Python and highlight the differences with other languages [10].

Results. Our findings indicate that the phenomenon of co-occurring TD is widely spread, i.e, more than 90% of Python files have been affected by at least two types of TD at the same time. Moreover, Design Debt issues are the most likely to co-occur with issues of similar nature, while Documentation Debt and Test Debt appear in the majority of the files. Test Debt issues co-occur the most frequently with other types, although the majority of such associations does not seem strong.

We also found that co-occurring issues tend to be introduced within the same week. However, after one of the co-occurring issues is removed, the second one survives for much longer (often around a year). The most notable exception to this relates to Design Debt: co-occurring Design Debt issues are likely to be removed together within a week.

Implications. Our findings are relevant to both researchers and practitioners. For the former, the results would be useful to develop tools to prioritize TD remediation and warn about potential future co-occurring issues. For example, the co-occurring relationship between Design Debt and Code Debt could be used to indicate design problems by using existing code smell detectors. For the latter, the findings can be used as guidelines to avoid the introduction of TD and consciously reduce co-occurring TD, which would in turn reduce the cost of software maintenance.

Structure of the paper. Section II discusses the study objectives, the research questions, and provides details regarding the data collection and analysis. Section III reports on the results of our study and Section IV discusses the results and their implications. Section V reports on the threats to the validity of our study and provides information about our dataset and replication package. After the discussion of related work in Section VI, Section VII concludes the paper and outlines the directions of future work.

II. Study Design

This empirical study was designed according to the guidelines of Runeson et al. [11] and is reported according to the Linear Analytic Structure [11].

A. Objectives and Research Questions

The goal of our study, described according to the Goal-Question-Metrics (GQM) approach [12], is to "*analyze* Apache software systems written in Python *for the purpose of* investigating various types of technical debt *with respect to* the co-occurrence of these types, *from the point of view of* software developers *in the context of* open source software". This objective is further refined into two research questions:

RQ1: Which types of technical debt frequently co-occur?

The goal of this research question is to analyze the types of technical debt that co-occur in the same file. The results can shed further light on understanding the 'domino effect' of particular types of associated debt, i.e., specific types of debt that usually lead to other types. This may draw developers' attention to specific types of TD that tend to result in even more TD, and thus extra complexity and maintenance effort.

RQ2: How long is the interval between the appearance and disappearance of co-occurring technical debt issues?

This research question aims at analyzing the average time between the appearance and disappearance of the co-occurrences. Such information would provide additional insights to developers, further supporting the prioritization of remediation. For example, a short time-span (on average) between different co-occurring issues may strongly indicate that one causes the introduction or removal of the other.

B. Case Selection

To perform our analysis, we considered Python projects from the ASF as subject systems. These have high quality and long-term stability since they are managed by self-selected teams of technical experts[3]. The ASF includes 48 Python projects of different domains, sizes (up to 1,000 KLOC), activity (up to 10K commits to the master branch) and number of files (up to 700) on GitHub. To select systems among them, we used two main inclusion criteria:

1) The project must show up on the Apache Projects List[4], which excludes Apache Incubator projects. Incubated projects are on a transition period to conform to Apache standards and, are therefore non-representative.
2) The project's main programming language must be Python, i.e., the largest number of files and source lines of code (SLOC) are written in Python.

Based on the criteria, we selected 20 Python projects[5], which include 3682 Python source files from the ASF. The projects have an average of 1410 commits (median: 187; max: 10942), 124 files (median: 32; max: 865) and 17K SLOC (median: 4K; max: 126K). The majority of projects have a long history of commits (at least 3 years), which allows following the co-occurrence of technical debt issues over an extensive period. The majority also has a large number of files, which mitigates potential threats to the external validity (i.e. how far the sample represents the population).

To analyze the evolution of project files, the historical information of the project is required. To enable the study of long-lived software systems and standardize the data collection, we decided to take equally-spaced snapshots. In particular, a period of one week was used to guarantee the inclusion

[2]According to the Tiobe Index, one of the best known indices of programming languages popularity, https://www.tiobe.com/tiobe-index/

[3]http://www.apache.org/foundation/how-it-works.html

[4]https://www.apache.org/index.html#projects-list, visited in Sep. 2019

[5]A detailed description of the projects can be found at https://github.com/jieshanshan/SEAA2020/blob/master/Projects.md

of sufficient revisions for each analyzed system. The same method was used in another research on TD evolution [13].

C. Variables

Each unit of analysis comprises the tuple:

<*file identification*; *snapshot time-stamp*; *TD information*>

The *file identification* comprises the *project name* and *file path*. The *snapshot time-stamp* regards the date in which the weekly snapshot was taken. Finally, *TD information* regards the amount of open issues in a particular file and snapshot. Section II-C1 elaborates on the types of TD collected, while Section II-C2 explains the data collection procedure.

1) Technical Debt Identification: To perform our study, as aforementioned, we use SonarQube as the tool to detect technical debt. SonarQube defines a set of rules to detect various types of technical debt and classifies them into four severity levels based on impact and likelihood: *blocker*, *critical*, *major* and *minor*. The *minor* issues are trivial and commonly perceived as having little to no relevance (e.g., *Lines should not end with trailing whitespaces*) and could, therefore, bias the results[6]. Thus, we limited the severity level to *blocker*, *critical* and *major*. During analysis, SonarQube creates a new issue when a piece of code breaks one of the predefined rules.

Moreover, issues related to some rules (e.g., *Syntax error*) are essentially interpreter errors instead of technical debt; thus, we also excluded such rules. We clarify that we maintain rules that can be associated with changes between Python2 and Python3 (e.g., *The "print" statement should not be used*). After removing issues of minor severity and related to interpreter errors, we are left with a total of 42 rules[7]. These rules are mapped to categories (e.g. complexity, duplicated code) and then to the five TD types[8], i.e., Design Debt, Code Debt, Test Debt, Documentation Debt and Defect Debt [10]. These types were independently derived by two different studies, i.e. by Alves et al. [2] and Li et al. [1]. The *TD information* variable for each unit of analysis is the count of issues for each of the 42 rules.

2) Data Collection: To perform our study, we started by cloning the GitHub repositories of the Python projects. Then, we wrote a script to: (1) extract the entire change history of each project; (2) group it by week, therefore defining the snapshots, and (3) submit the snapshots to SonarQube in chronological order. The weekly commits were then analyzed by SonarQube to identify the technical debt issues.

Finally, we use a second script to obtain all the information for each issue, i.e., which rule it breaks, when it appeared and disappeared in the file and the name of the file. With this information, we calculate the variables the comprise the TD information of each unit of analysis, i.e., the TD of each file for each snapshot.

Overall, we detected more than 23K issues related to 42 different rules of technical debt from over 45K commits of selected 20 Python projects.

D. Data Analysis

In this section, we first explain a method that is key to our analysis, namely *Association Rule Mining*. Subsequently we describe the steps necessary to analyze the collected data and answer the proposed research questions.

1) Association Rule Mining: This method is used for detecting an association between different objects in a set, by finding frequent patterns in a transaction database, relational database or any other information repository. In this paper, we perform association rule mining based on a well-known machine learning algorithm, namely Apriori [14]. The same method is also used by Palomba et al. [5]. Following the original definition by Agrawal et al. [14], we let $TD = \{td_1, td_2, \ldots, td_n\}$ be a set of n binary attributes composed by all types of technical debt and each file contains a subset of the types of *TD*.

In our study, the direction of the relation is defined as an implication of the form:

$$td_i \rightarrow td_j, \text{where } td_i, td_j \in TD, \text{ and } i \neq j, \tag{1}$$

to represent the relationship that td_i appears before td_j in the same source file.

To answer each RQ, we introduce the following formula to measure association between different types of technical debt:

$$\texttt{confidence}_{td_i \rightarrow td_j} = \frac{\texttt{support}(td_i \wedge td_j)}{\texttt{support}(td_i)} \tag{2}$$

where $\texttt{support}(td_i \wedge td_j)$ is the probability of both td_i and td_j appearing in the same file and $\texttt{support}(td_i)$ is the percentage of files that contain td_i. This formula represents the likelihood of td_j appearing when td_i already exists in the same file, expressed as $td_i \rightarrow td_j$.

The drawback of this method in calculating the percentage of co-occurrences of td_i and td_j is that it might misrepresent the relationship, as it only accounts for how popular the issue td_i is but does not consider td_j. If the rule td_j is also appearing frequently in general, there will be a higher chance that a file containing td_i will also contain td_j, thus impacting on the value and accuracy of the relationship between co-occurring technical debt.

To account for the prevalence of co-occurrences, we use the following formula:

$$\texttt{lift}_{td_i \rightarrow td_j} = \frac{\texttt{support}(td_i \wedge td_j)}{\texttt{support}(td_i) \times \texttt{support}(td_j)}, \tag{3}$$

which can be interpreted as the probability of both td_i and td_j co-occurring in the same file while controlling for the popularity of both td_i and td_j. The interpretation of the results of this formula is as follows:

[6]https://docs.sonarqube.org/latest/user-guide/rules/, visited in Dec. 2019

[7]The detailed description of the rules can be found at https://github.com/jieshanshan/SEAA2020/blob/master/TD_Rules.csv

[8]The mapping of rules to categories and types is explained at https://github.com/jieshanshan/SEAA2020/blob/master/RuleType_map.pdf

- if $\texttt{lift}_{td_i \to td_j} = 1$, it would imply that the probability of occurrence of td_i and td_j are independent of each other, thus no rule can be drawn involving them.
- if $\texttt{lift}_{td_i \to td_j} > 1$, the value gives the degree of interdependence between td_i and td_j, and makes that relationship potentially useful for predicting the consequent technical debt in the future. Greater lift values indicate stronger associations.
- if $\texttt{lift}_{td_i \to td_j} < 1$, it means that presence of td_i has negative effect on presence of td_j, and vice versa.

2) Analysis Procedure: To answer **RQ1**, we use the information of the evolution per issue to find all the dates on which rule violations were introduced or removed, henceforth referred to as *debt-changed-date*. After that, we calculate the percentage for each TD rule on the 20 analyzed projects per *debt-changed-date* and use the average value $\texttt{support}(td_i)$ to represent how popular td_i is during the evolution of these projects.

Then, we use the percentage of files that contain both td_i and td_j to divide the percentage of files that only contain td_i to calculate $\texttt{confidence}_{td_i \to td_j}$ for each project on its *debt-changed-dates*. Similarly, we also get the average value as the final result for $\texttt{confidence}_{td_i \to td_j}$ to investigate how often the existence of one rule td_j relates to the occurrence of another rule td_i in the same file.

As $\texttt{confidence}_{td_i \to td_j}$ shows how popular a co-occurring relationship between td_i and td_j is, a low value of $\texttt{confidence}_{td_i \to td_j}$ means that a co-occurrence $td_i \to td_j$ is likely to be occasional and meaningless, or one of the debt rules (i.e., td_i or td_j) does not appear frequently during the evolution of Python projects. To avoid the analysis of meaningless relationships, we choose the top ten percent of the total $\texttt{confidence}_{td_i \to td_j}$ values.

Finally, we investigate how often the occurrence of one rule td_i in a source file is related to the introduction of another rule of technical debt td_j. It is worth noting that the previous results show how popular the co-occurring rules are. However, we still cannot be sure that there is a strong relationship between them, especially when td_j appears much more frequently than the other rules. In that case, we do not have strong evidence that the introduction of td_j is related to the introduction of td_i. To handle this research question, we select the valid co-occurrences with $\texttt{lift}_{td_i \to td_j} > 1$.

For **RQ2**, we analyze the pairs of co-occurring TD detected in **RQ1**. For each pair, we use all reported issues to calculate and analyze the interval of introduction and removal between two co-occurring issues found in **RQ1**. These variables are measured as the number of days between the moments when they are introduced (Formula 4) and removed (Formula 5) in the file:

$$\texttt{intro-interval}_{td_i \to td_j} = \texttt{intro}_{td_i} - \texttt{intro}_{td_j} \quad (4)$$

$$\texttt{remov-interval}_{td_i \to td_j} = \texttt{remov}_{td_i} - \texttt{remov}_{td_j} \quad (5)$$

note that $\texttt{intro-interval}_{td_i \to td_j}$ differs from $\texttt{intro-interval}_{td_j \to td_i}$, and if td_i is introduced before td_j, the value of $\texttt{intro-interval}_{td_i \to td_j}$ is negative. For the removal of co-occurring issues, if only one issue (e.g., td_j) is removed and the other co-occurring issue (td_i) still exits in the file, we regard the removal time of the latter as an infinite value, i.e., the interval of removal is infinite ($\texttt{remov-interval}_{td_i \to td_j} \to \infty$). As a reminder, we are interested in both issues being removed instead of just one; this is because we are interested in associating the removal of one to the other, especially if removing one causes the removal of the other.

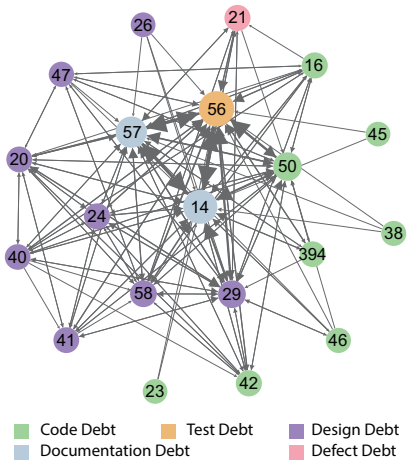

Fig. 1. Network of top 10% co-occurring rule violations

III. Results

A. RQ1: Which types of technical debt frequently co-occur?

We calculated the number of TD types (i.e., Design Debt, Documentation Debt, Test Debt, Code Debt, Defect Debt) that affected each file at the same period. We found that 90.3% (i.e., 3324 out of 3682) of the files were affected by at least two debt types. This signifies that Python files are predominantly afflicted by multiple TD issues.

To examine this phenomenon in more depth, we look at the numerous rules for the different TD types (see Section II-C). Specifically, in the following we compare the confidence values of every pair of co-occurring rules. For that, we organized the confidence values of the 1722 pairs in bins of size 0.02. We found that the majority of the confidence values (72.6%) are under 0.02, i.e., **most of the associations tend to be occasional and meaningless**.

As discussed in Section II-D2, we choose the top ten percent of the total $\texttt{confidence}_{td_i \to td_j}$ values. Figure 1 shows a network describing those selected 172 top co-occurring associations (as aforementioned, the ID, types and descriptions of the rules can be found online[9]). Each node represents one TD rule, while the node label is marked with the rule ID number. The nodes are color-coded according to their TD type. Larger nodes denote higher popularity (i.e., chance of appearing in a file). The edges are directed according to the direction of the co-occurrence. From this figure, one can

[9]https://github.com/jieshanshan/SEAA2020

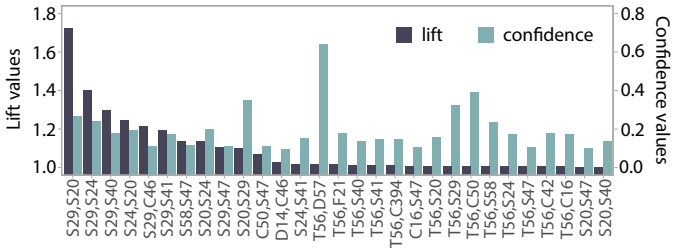

Fig. 2. Distribution of the possibility of co-occurrence and lift values for the violations of 29 pairs of rules

TABLE I
MAPPING OF HIGHLIGHTED RULES TO TYPES OF TECHNICAL DEBT

R^i	Definition	$S\%^a$
T56	Lines should have sufficient coverage by tests	98.2
D14	Docstrings should be defined	87.8
D57	Source files should have a sufficient density of comment lines	62.8
C50	Lines should not be too long	39.0
S29	Cognitive Complexity of functions should not be too high	32.0
S58	Source files should not have any duplicated blocks	23.4
C42	Function names should comply with a naming convention	17.7
F21	The "print" statement should not be used	17.7
S24	Control flow statements should not be nested too deeply	17.2
C16	Sections of code should not be "commented out"	17.1
S20	Functions should not be too complex	15.3
S41	Collapsible "if" statements should be merged	14.6
C394	File should not use "import*"	14.2
S40	Functions should not contain too many return statements	13.5
S47	Files should not have too many lines of code	10.1
C46	Functions, methods and lambdas should not have too many parameters	9.0

^{i} ID number of the rule
(S=design debt, C=code debt, D=documentation debt, F=defect debt, T=test debt)
^{a} `support` values, i.e., the percentage of files that contain the rule

observe that some of the more central nodes regard Design Debt, while Documentation and Test Debt are also very central but with two and one nodes respectively.

To investigate how often the occurrence of td_i in a source file is related to the appearance of another rule td_j, we choose the co-occurrences with $\texttt{lift}_{td_i \to td_j} > 1$. Table I reports detailed information on the top 16 rules involved in these selected associations, denoting the type, ID number, definition and percentage of files that contain it. Figure 2 reports a bar chart comparing the lift values and the probability of the co-occurrences. We do not observe a correlation between them. This is mostly because the probability of the co-occurrence is greatly affected by the popularity of each rule, but lift value is not. In short, the likelihood of $\texttt{confidence}_{td_i \to td_j}$ does not reflect on the strength of the relationship and, thus, the examination of $\texttt{lift}_{td_i \to td_j}$ helps to normalize the co-occurrence by the popularity of both co-occurring rules.

To illustrate this through an example from Figure 2: $\texttt{confidence}_{T56 \to D57}$ is the most likely association since it displays the highest value (63.9%) among the 37 co-occurrence associations. However, $\texttt{lift}_{T56 \to D57}$ is close to the median value, which may denote that both T56 and D57 appear in the majority of Python files. The percentage of files that contain one of these two rules is shown in Table I. Therefore, it has indeed a high chance of simultaneously occurring in the same file.

The top four co-occurrences are between Design Debt issues regarding complexity, i.e., *cyclomatic complexity* (S20), *cognitive complexity* (S29) and *spaghetti code* (S24). As shown in Figure 2, $\texttt{confidence}_{S20 \to S29}$ is the highest one, i.e., 35.1% of files affected by S20 (*cyclomatic complexity*) are also affected by S29 (*cognitive complexity*), followed by $\texttt{confidence}_{S29 \to S20}$ (26.3%). This indicates that **cyclomatic complexity and cognitive complexity are strongly intertwined** during the evolution of Python files. In other words increase in cognitive complexity triggers increase in cyclomatic complexity and vice versa. Furthermore, the results suggest that **cyclomatic complexity (S20) and spaghetti code (S24) have a similar impact on each other**, i.e., similar probability of co-occurrence.

Regarding Test Debt (T56) we observed that it **tends to co-occur with other rule violations and appears in almost all files throughout their evolution**. However, all of the lift values that are related to Test Debt are close to 1, i.e., **the probability of insufficient test coverage for introducing other rule violations is reduced**.

Regarding Defect Debt, one rule (F21) out of the 15 available is involved in the frequently co-occurring associations. Moreover, the lift values between Defect Debt and Test Debt are higher than the other rules related to Design Debt and Code Debt in Figure 2. These observations show an indication that **incompleteness of software testing could be associated with an increase in Defect Debt**. In addition, issues related to F21 are caused by the update of the Python interpreter (from Python2 to Python3). Thus, the increase of test coverage may greatly support maintenance activities when the interpreter is updated.

Concerning the relationship between Test Debt and Code Debt or Design Debt, all of the lift values between them are near 1.01. Thus, there seems to be **no obvious relationship between the introduction of Test Debt and the increase in Code Debt and Design Debt**. In addition to rule T56, rule S29 (*Cognitive Complexity of functions should not be too high*), has the second largest number of co-occurrences. However, compared with T56, the lift values are higher than the others. Figure 2 shows that 70% of the top ten lift values are related to S29. Moreover, the majority of the other rules involved in these associations are also related to Design Debt, only two of them are related to Code Debt. Such evidence suggests strongly that **Design Debt issues are often associated with more problems that can be of similar nature (i.e. other Design Debt issues) or that can be caused by the same design decisions**.

B. RQ2: How long is the interval between the appearance and disappearance of co-occurring technical debt issues?

To answer this research question, we calculated the interval values ($\texttt{intro-interval}_{td_i \to td_j}$ and $\texttt{remov-interval}_{td_i \to td_j}$) for each pair of the frequently co-occurring associations found in RQ1 (Section III-A).

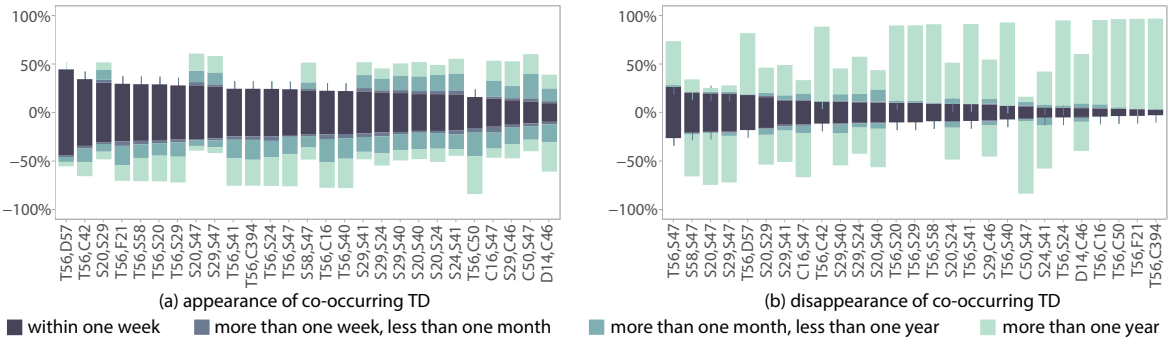

Fig. 3. Distribution of the possibility of co-occurrence and lift values for the violations of 27 pairs of rules

Note that if td_i is introduced before td_j, the value of $\texttt{intro-interval}_{td_i \to td_j}$ is negative. Similarly, if td_i is removed before td_j, the value of $\texttt{remov-interval}_{td_i \to td_j}$ is negative.

Figure 3(a) and Figure 3(b) show diverging stacked bar charts to visualize the percentages that describe the distribution of intervals for the different pairs of co-occurring technical debt issues. The percentages show how often an issue td_i was introduced (Figure 3(a)) or removed (Figure 3(b)) before an issue td_j within a certain time-frame (i.e., one week, one month, one year and after one year). The time-frames are represented by the different shades of colors, ranging from "within one week", to "more than one year".

The main finding based on Figures 3(a) and Figure 3(b) is that **if co-occurring issues are not introduced within one week, then there is a negligible chance of them appearing within one month**; in most cases they will appear after at least one year. However, once introduced, we notice that **co-occurring issues tend not to be removed within weeks or months, but will likely survive at least one year** in the file. Combining these two findings, TD co-occurrence seems to happen either very quickly or not at all; but once it does happen, it is hard to get rid of. Furthermore, **co-occurring Design Debt issues seem to be repaid in a shorter time** as shown in Figure 3(b).

IV. Discussion

A. Interpretation of Results

The phenomenon of co-occurring TD is widely spread in Python projects; this confirms a similar observation regarding Java projects [5]. Moreover, all the Design Debt rules shown in Table I are not Python-specific and have also been detected in projects written in other programming languages, such as Java and JavaScript[10]. This indicates that these particular results do not apply to Python only. For example, we observe that rule S58 (*duplicated blocks*) co-occurs with S47 (*long file*), which is similar to what Yamashita et al. [15] observed for Java, i.e., duplications appear together with smells related to size. Thus, **some co-occurring TD issues may apply also to Java.**

However, the rest of our results are limited to Python and can aid practitioners and researchers specifically for this language. To begin with, there tends to be a strong association between Design Debt issues and Code Debt issues. This result is supported by the high lift values observed in the investigation of RQ1 and the shorter removal intervals observed in the investigation of RQ2. It indicates that **investigating issues at code level can support assessment and reasoning about the design of a software-intensive system**.

Comparing the network in Figure 1 and the corresponding support values in Table I, we notice that Design Debt issues are strongly interconnected. However, this is not uniform across all projects. For example, issues related to cyclomatic complexity (S20) and spaghetti code (S24) co-occur most frequently in a software development collaboration tool (BLOODHOUND), but those co-occurrences are less likely to appear in projects for messaging APIs or test suites (e.g., QPID). Thus, **projects in different domains show differences regarding which TD issues co-occur**.

Moreover, Design Debt issues not only seem to be strongly associated with one another (RQ1), but they also tend to be removed together and quickly (RQ2). For example, issues related to long files (S47) tend to be resolved together with other issues the fastest; this is probably because files are shortened when undertaking refactoring activities to address other (co-occurring) issues. These findings corroborate earlier findings [5] that **Design Debt issues are generally removed together as a consequence of maintenance and evolution activities**.

We also found that, although Test Debt tends to co-occur with issues regarding other TD types, it has a lesser impact on the increase of Code Debt or Design Debt. These results are not in full accordance with related work, which suggest that Test Debt has a strong negative influence on maintenance [16]. A possible reason is that only one rule related to Test Debt was involved in our study. To further investigate this relationship, we calculated the Spearman correlation between the percentage of files that contain both types of debt (Test and Design Debt) and the percentage of files containing Design Debt only (see the corresponding values in Table I). We found the correlation to be positively strong[11] ($\rho = 0.98$), which

[10]https://rules.sonarsource.com/

[11]We interpret the correlation coefficient according to Cohen [17], i.e., strong correlation when $0.5 \leq |\rho| \leq 1$.

further confirms that **Test Debt has a lesser impact on the increase of Design Debt**.

Comparing the findings from the appearance and disappearance of co-occurring TD (Figures 3(a) and 3(b)), the percentages of co-occurring rule violations that are introduced within one week are much higher than those that are removed within one week, especially for Test Debt. Moreover, Test Debt is always removed later than others. These observations are possibly related to adopted practices, or lack thereof, as we were able to identify evidence of test-driven development in only one project (ALLURA). This indicates that, since it always appears earlier than the others, **Test Debt seems easier to incur, but also more difficult to resolve, possibly because it is perceived as less important.**

B. Implications for Researchers and Practitioners

Based on the interpretation of our results, we highlight that:
(1) Design Debt issues may be associated with more similar problems; thus, practitioners should target them in earlier refactoring activities. On the contrary, fixing Test Debt can be postponed since it has a lesser impact on increasing the probability of introducing other TD issues and tends to be harder to resolve. However, one should be careful with such trade-offs as refactorings may lead to bugs and, without appropriate tests, also to catastrophic results.
(2) The findings might help practitioners to estimate the risk level of potential TD issues and make decisions over design problems by using code-level insights. For example, the accumulation of issues such as cognitive complexity and code duplication in components could raise the risk of specific design problems emerging in the near future.
(3) Researchers can use our results to guide the development of theoretical frameworks and associated research tools, as well as a basis for future studies. In particular, the results of frequently co-occurring TD can be used as an input to calculate TD indices that integrate metrics from individual issues with their co-occurrence information. In addition, TD tools can improve the prioritization of TD remediation by calibrating the weights for the different TD issues. For example, Design Debt issues can be assigned higher weights since they are more interconnected and repaid in a shorter time.
(4) Our findings can also be used to warn developers about potential future co-occurring issues. For example, developers could be cautioned that incompleteness of software testing could be an indicator of increasing Defect Debt, especially Defect Debt issues caused by the update of the interpreter.

V. THREATS TO VALIDITY

In the following, we discuss the threats to construct and external validity of the reported study, as well as reliability threats. We note that we do not analyze internal validity, since we do not seek to establish causal relations but to study the associations between various TD issues.

Construct validity pertains to the connection between the research questions and the objects of study. In this regard, the result of this study relies on SonarQube to detect TD issues. Although the tool has been widely used in both industry and academia, our interpretation of TD is limited to the tool's capabilities. Different tools could use different strategies to detect TD, which might lead to variations in the TD issues and, consequently, potential co-occurrence associations.

Another threat is related to how we analyze the evolution of projects. In particular, we analyze the change history on a weekly basis and, thus, the introduction and removal time might have a maximum error of one week. Moreover, projects may receive new commits at different rates, which may affect the time-frame between the introduction or removal of co-occurring issues in different projects. To mitigate potential bias we use the Apriori method to identify the associations.

The analyses reported in this paper were performed at file level, which may be affected by the high diffuseness of rules from a certain TD type (e.g. Test Debt), increasing the probability of co-occurring issues. Other studies have also performed analyses at file level to investigate co-occurrence [7], [15] but, unlike them, we introduced the lift value to further control for the popularity of co-occurring issues.

External validity concerns threats to the generalizability of our findings. The main threat here comes from the fact that we analyzed the evolution of 3862 files from 20 projects that are limited to the ASF and Python systems. Despite the credibility of the Apache Foundation and the diversity of its Python projects, our results may not fully represent the entire population of non-trivial Python projects. Furthermore, the set of rules considered in this study is not exhaustive and does not portray the complete set of TD related issues that may affect Python source code.

We emphasize that the size of the dataset is also limited by the use of SonarQube and its multi-version analysis: collecting the data required for this analysis can take a long time. In our case, analyzing 20 systems took more than two months of work on an Intel Core i7-5500U PC with 8GB of RAM.

Reliability threats were addressed by involving at least two researchers in both data collection and analysis. Also, samples of analyses output at the different steps were manually inspected for irregularities and alignment with the proposed study design.

Finally, we created an online repository with instructions and scripts to collect the same data that we used in the study[12]. The repository helps with setting up the necessary environment, i.e., tools and configuration. The provided Jupyter[13] notebook guides the procedure from acquiring the Git repositories, extracting the weekly snapshots, and submitting them to SonarQube for analysis.

VI. RELATED WORK

In recent years, the research community has extensively analyzed the effect of individual instances of TD [4], while several studies also focused on the correlation between different items of technical debt. Some results indicate that code smells that

[12]https://github.com/jieshanshan/SEAA2020
[13]https://jupyter.org/

co-occur in the same source file can intensify negative effects on software maintenance [7], [15]. Palomba et al. [5] investigated six pairs of frequently co-occurring instances among 13 code smells, and found that code smells are generally removed together as a consequence of maintenance. Fontana et al. [18] found that code smells co-occur and interact in many ways, which is confirmed in our study.

Several studies also focused on the association between different debt types. Abbes et al. [19] found that the combination of two design smells, namely God Class and Spaghetti Code, significantly reduces program comprehension when compared to only one. The results suggest that developers could deal with one design smell but the combination of different design smells should be avoided through detection and refactoring. Spadini et al. [16] investigated the relationship of six test smells and their co-occurrence with software quality. They found that detecting test smells is important for the effectiveness of defect detection. Bavota et al. [3] studied the relationship between test smells and their effects on software maintenance, observing that these smells are highly diffused [3]. However, we did not confirm that Test Debt has a strong negative impact on issues related to software maintenance, i.e., Code Debt or Design Debt.

Unlike the aforementioned studies, which focused on one or two debt types only, we analyzed the co-occurrence between five types of debt.

VII. Conclusion and Future Work

This paper reports on an empirical study that investigated the relationship between co-occurring technical debt types. We performed our study by analyzing the multi-annual evolution of TD in 20 Python projects (approx. 28,200 commits) from the ASF and identifying TD based on 42 rules defined in SonarQube. Moreover, we mined co-occurring TD based on a well-known machine learning algorithm, namely Apriori, in 3862 files.

We found that the phenomenon of co-occurring TD is highly spread, i.e., more than 90% of Python files have been affected by at least two types of TD at the same time. In particular, from the 1722 possible associations between rules, 48 pairs frequently co-occurred in files, from which 27 are more strongly associated with each other. Among them, Design Debt issues more commonly co-occur with issues of similar nature, while Documentation Debt and Test Debt are the most prevalent in the studied projects.

Furthermore, we found that co-occurring issues tend to be introduced within the same week. However, after one of the co-occurring issues is removed, the second one survives for much longer (often around a year). Moreover, co-occurring Design Debt issues seem to be repaid in a shorter time, while most of the Test Debt issues are introduced before the other debt types but it is also the last to be repaid. These findings may indicate that developers are more aware of Design Debt compared to the other types, and that Test Debt may be regarded as a difficult type of debt or with less priority.

In future work, we plan to investigate the relationship between the habits of developers and co-occurring TD. We also plan to develop a tool that can use our results to warn developers about potential issues and help them to make more informed decisions prior to refactoring activities.

References

[1] Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management," *J. of Syst. and Software*, vol. 101, no. C, pp. 193–220, Mar. 2015.

[2] N. Alves, L. F. Ribeiro, V. Caires, T. Mendes, and R. Spínola, "Towards an ontology of terms on technical debt," in *Proc. 6th Int. Workshop on Managing Technical Debt*, Sep. 2014, pp. 1–7.

[3] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "Are test smells really harmful? an empirical study," *Empirical Software Eng.*, vol. 20, no. 4, pp. 1052–1094, 2015.

[4] D. I. K. Sjøberg, A. Yamashita, B. C. D. Anda, A. Mockus, and T. Dybå, "Quantifying the effect of code smells on maintenance effort," *IEEE Trans. Softw. Eng.*, vol. 39, no. 8, pp. 1144–1156, Aug. 2013.

[5] F. Palomba, G. Bavota, M. D. Penta, F. Fasano, R. Oliveto, and A. D. Lucia, "A large-scale empirical study on the lifecycle of code smell co-occurrences," *Inform. and Software Technology*, vol. 99, pp. 1–10, 2018.

[6] ——, "On the diffuseness and the impact on maintainability of code smells: A large scale empirical investigation," *Empirical Software Eng.*, vol. 23, no. 3, pp. 1188–1221, Jun. 2018.

[7] A. Yamashita and L. Moonen, "Exploring the impact of inter-smell relations on software maintainability: An empirical study," in *Proc. 35th Int. Conf. Software Eng. (ICSE '13)*, May 2013, pp. 682–691.

[8] M. R. Dale and C. Izurieta, "Impacts of design pattern decay on system quality," in *Proc. 8th ACM/IEEE Int. Symp. Empirical Software Eng. and Measurement (ESEM '14)*. Torino, Italy: ACM, 2014, pp. 37:1–37:4.

[9] B. Wang, L. Chen, W. Ma, Z. Chen, and B. Xu, "An empirical study on the impact of python dynamic features on change-proneness," in *Proc. 27th Int. Conf. Software Eng. and Knowledge Eng. (SEKE '15)*, Pittsburgh, PA, USA, 2015, pp. 134–139.

[10] J. Tan, D. Feitosa, and P. Avgeriou, "An empirical study on self-fixed technical debt," in *Proc. 3rd IEEE/ACM Int. Conf. Technical Debt (TechDebt '20)*. ACM, Oct. 2020, pp. 1–12.

[11] P. Runeson, M. Host, A. Rainer, and B. Regnell, *Case Study Research in Software Engineering: Guidelines and Examples*. Wiley Blackwell, 2012.

[12] R. van Solingen, V. Basili, G. Caldiera, and H. D. Rombach, "Goal Question Metric (GQM) approach," in *Encyclopedia of Software Eng.* Hoboken, NJ, USA: John Wiley & Sons, Inc., Jan. 2002, pp. 528–532.

[13] G. Digkas, M. Lungu, P. Avgeriou, A. Chatzigeorgiou, and A. Ampatzoglou, "How do developers fix issues and pay back technical debt in the apache ecosystem?" in *Proc. IEEE 25th Int. Conf. Software Analysis, Evolution and Reengineering (SANER '18)*, Mar. 2018, pp. 153–163.

[14] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," in *Proc. 1993 ACM SIGMOD Int. Conf. Management of Data (SIGMOD '93)*. Washington, D.C., USA: ACM, 1993, pp. 207–216.

[15] A. Yamashita, M. Zanoni, F. A. Fontana, and B. Walter, "Inter-smell relations in industrial and open source systems: A replication and comparative analysis," in *Proc. 2015 IEEE Int. Conf. Software Maintenance and Evolution (ICSME '15)*, Sep. 2015, pp. 121–130.

[16] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli, "On the relation of test smells to software code quality," in *Proc. 2018 IEEE Int. Conf. Software Maintenance and Evolution (ICSME '18)*, Sep. 2018, pp. 1–12.

[17] J. Cohen, *Statistical power analysis for the behavioral sciences*. Lawrence Erlbaum Associates, 1988.

[18] F. A. Fontana, V. Ferme, and M. Zanoni, "Towards assessing software architecture quality by exploiting code smell relations," in *Proc. IEEE/ACM 2nd Int. Workshop on Software Architecture and Metrics*, May 2015, pp. 1–7.

[19] M. Abbes, F. Khomh, Y. Gueheneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *Proc. 15th European Conf. Software Maintenance and Reengineering*, Mar. 2011, pp. 181–190.