Impact of Opportunistic Reuse Practices to Technical Debt

Capilla, Rafael

IEEE Computer Society
2021

Capilla , R , Mikkonen , T , Carrillo , C , Fontana , F A , Pigazzini , I & Lenarduzzi , V 2021 , Impact of Opportunistic Reuse Practices to Technical Debt . in 2021 IEEE/ACM INTERNATIONAL CONFERENCE ON TECHNICAL DEBT (TECHDEBT 2021) . IEEE Computer Society , pp. 16-25 , 4th IEEE/ACM International Conference on Technical Debt (TechDebt) , 22/05/2021 . https://doi.org/10.1109/TechDebt52882.2021.00011

http://hdl.handle.net/10138/341457
https://doi.org/10.1109/TechDebt52882.2021.00011

acceptedVersion

Impact of Opportunistic Reuse Practices to Technical Debt

Rafael Capilla
Rey Juan Carlos University
Madrid, Spain
rafael.capilla@urjc.es

Tommi Mikkonen
University of Helsinki
Helsinki, Finland
tommi.mikkonen@helsinki.fi

Carlos Carrillo
Technical University of Madrid
Madrid, Spain
carlos.carrillo@upm.es

Francesca Arcelli Fontana
University of Milano-Bicocca
Milano, Italy
francesca.arcelli@unimib.it

Ilaria Pigazzini
University of Milano-Bicocca
Milano, Italy
i.pigazzini@campus.unimib.it

Valentina Lenarduzzi
LUT University
Lahti, Finland
valentina.lenarduzzi@lut.fi

***Abstract*—Technical debt (TD) has been recognized as an important quality problem for both software architecture and code. The evolution of TD techniques over the past years has led to a number of research and commercial tools. In addition, the increasing trend of opportunistic reuse (as opposed to systematic reuse), where developers reuse code assets in popular repositories, is changing the way components are selected and integrated into existing systems. However, reusing software opportunistically can lead to a loss of quality and induce TD, especially when the architecture is changed in the process. However, to the best of our knowledge, no studies have investigated the impact of opportunistic reuse in TD. In this paper, we carry out an exploratory study to investigate to what extent reusing components opportunistically negatively affects the quality of systems. We use one commercial and one research tool to analyze the TD ratios of three case systems, before and after opportunistically extending them with open-source software.**

***Index Terms*—Technical debt, opportunistic reuse, architectural debt**

I. Introduction

Managing technical debt [1] has become a challenging research area for software engineers to keep the quality of systems under control and ease software maintenance tasks. The evolution of technical debt (TD) approaches since several years has produced significant advances in terms of metrics and tools to estimate the different forms of debt, with particular attention paid to architecture and code debt [2]. Recent studies [3] have shed light on modern sources of technical debt in software-intensive systems, on how software organizations cope with TD symptoms, and on ways and strategies to handle and fix the issues (e.g. bugs, smells) adopted by the many forms of technical debt [4] [5].

One of the factors that may influence the estimation of technical debt is the reuse of third-party components and their integration into an existing system. As reuse practices may have strong implications in the architecture (e.g. when a platform or a protocol is replaced), we need to estimate the impact on architectural debt derived from the reuse of software components and when new functionality is added. As Martin Griss [6] stated in 2015, "*software reuse it isn't what it use to be*", wishing to highlight the evolution of reuse practices from feature reuse (i.e. mainly used in software product line approaches) to reuse in sourcing and crowd models. Nowadays, many of the systematic reuse approaches have been replaced in favor of opportunistic reuse, where developers search for reusable components in key repositories, such as GitHub, Bitbucket, or Gitlab. In many cases, selecting the right open source components is not easy as many factors, such as licensing, popularity, low code quality affecting technical debt, or the build process after integration, may complicate the selection process [7]. In addition, the impact of reusing and integrating third-party components may affect important quality attributes such as safety, integrability, and reliability among others. In other cases, upgradeability plays an important role in the selection of a particular component versus other options.

Some recent studies [8] describe a reusability index based on several metrics to quantify reuse of reusable assets, but the work does not describe a connection to technical debt when reusing components. As to the best of our knowledge there are no works exploring the impact and the connection opportunistic reuse plays in TD, in this research work we analyze how this opportunistic reuse trend affects the technical debt ratios in several open-source projects and we analyze such TD ratios before and after reusing third-party components, and what are the implications for the software architecture. More specifically, we used SonarQube[1] and Arcan [9] tools to analyze the code and architectural debt ratios in three different applications to investigate the effects of reusing functionality in open source repositories.

The rest of this work is structured as follows. In Section II we describe related work concerning the different legs of this research. In Section III we provide an overview of SonarQube and Arcan tools used in this study. Section IV outlines the case study design. The results are described in Section V and Section VI discusses our findings. In Section VII we discuss

[1]https://www.sonarqube.org/

the threats to validity of our work, and in Section VIII we draw some final conclusions.

II. Related Work

In this section, we will describe the most relevant related work, covering opportunistic reuse practices and the role of technical estimators for software quality analysis.

A. Opportunistic Reuse

It has traditionally been assumed that software reuse requires solid practices and principles to foster. For instance, Morisio et al. [10] conclude that successes in software reuse are achieved due to: commonality among applications, management committed to introducing reuse processes, modifying non-reuse processes, and addressing human factors. Similarly, Griss [11] asserts that architecture, process, organization, culture, management, and other non-technical factors are usually more critical for reuse than the use of a particular technology, and Jalender et al. [12] claim that organization, processes, and technical expertise are the prerequisites of successful software reuse. Tracz [13] mentions the concept of unplanned reuse (e.g. ad-hoc reuse) which occurs frequently in the software community but considered an error-prone and time-consuming task compared to software initially designed for reuse.

Today, modern forms of ad-hoc reuse (i.e. equivalent to the notion of opportunistic reuse) dig for reusable components into open-source repositories. However, with the excessive amount of open-source projects, the chances to find reusable assets are many. Then, when a partial piece of code is identified – an obvious key condition for reuse [14] – it can be opportunistically matched with other pieces of software until a satisfactory version of a program emerges. In contrast to the process-centric approach advocated by systematic reuse, such opportunistic reuse seemingly takes place in an ad-hoc fashion, scavenging whatever artifacts are found [15].

Strategies for selecting open source components have already been proposed [7], covering issues such as functionality, licensing, and popularity. Furthermore, open-source communities have made it more systematic to decide which subsystems and projects to reuse, resulting in various models (e.g., [16], [17], [18], [19]) that aim at describing the maturity of the open-source software. While there are subtle differences in the models [20], they share similar concerns [7], such as functionality, licensing, popularity, code quality, and community vitality.

Fisher et al. [21] have described clone-and-own techniques as a suitable form of systematic reuse being adopted in recent years. The authors further propose a methodology that can be applied at the level of variability models, the rationale being that clone-and-own techniques commonly lack a systematic process. Moreover, a recent survey [22] investigates different forms of reuse from past practices to current ones. Unfortunately, while the authors provide an insight into how practitioners do reuse, an in-depth analysis is missing regarding which of these practices are carried out opportunistically. Another recent work by Ali et al. [23] presents a hybrid DevOps process that follows systematic reuse-based software development and management process. The goal of the work is to reduce the effort and cost of reuse, based on information retrieval techniques. Finally, two recent papers [15], [24] highlighted the role of opportunistic reuse practices and their impact in architecture as well and how this trend is confirmed by practitioners and developers through a survey.

B. Technical Debt Estimators

There are many ways to estimate technical debt in code and architecture. Most tools used to estimate technical debt (TD) ratios rely on a combination of metrics to analyze statically code and architecture violations and map these to quality properties. Among these metrics, we can find a recent study [25] that investigates the relationships between architecture and design smells in several open-source repositories. Architectural smells, such as those described in [26], [27] are an important source of TD that must be estimated to detect possible violations of design principles (e.g. cyclic dependencies, ambiguous interface smells), and minimize the architectural debt [28], [29].

Nowadays, research tools like ARCADE [30] estimate how an architecture decays through the detection of architecture smells as indicators of TD symptoms. Designite is a commercial tool able to detect a large number of smells at implementation, design, and architectural level [31]. Also, few tools can detect the architectural debt index (ADI). Arcan [9] is an academic tool developed in this direction. The index provided by Arcan has been studied by Roveda et al. [32], and the authors provide a preliminary comparison between ADI and the Technical Debt Index computed by SonarQube. According to the different technical debt indexes provided by tools, a comparison of the indexes [33] and a comparison of these tools according to different TD features [34] have been explored in the literature. Other authors like Wu et al. [35] studied architectural debt indexes through a Standard Architecture Index including structure-related and global measures regarding both source code and software models. Finally, Verdecchia et al. [36] proposed a step-by-step method to build architectural debt indexes based on architectural violations that can be identified through static analysis rules.

Code-level TD has been investigated considering different strategies [37], [38], and ways to measure it [39], [40]. Code-level TD can be detected by different automated static analysis tools ($ASAT_s$) [34]. SonarQube is one of the $ASAT_s$ more adopted by developers in industry [41], [34]. However, few works estimate technical debt using SonarQube rules, focusing the change- and fault-proneness [42], [43]. Other works investigate the diffuseness of Technical Debt measured by SonarQube [44], [40]. SonarQube TD items detected as class level, have a negative influence increasing change-proneness [39], [41]. Considering the different types and severity assigned by SonarQube to TD items, there is no significant difference between the clean and infected classes. All the TD items have a statistically significant yet small effect on change-proneness. However, all the TD items classified as *Code Smell* affect change-proneness, even if their impact

on the change-proneness is low. Considering fault-proneness, there is no significant difference among the TD items that SonarQube claims to increase fault-proneness (i.e. Bugs), only one out of 36 has a limited effect, and 26 hardly never led to a failure. Unexpectedly, all the remaining *Bugs* resulted in a slight increase in the change-proneness instead [41]. Moreover, by removing any TD items, developers can prevent up to 20% of faults in the source code [42]. Moreover, the largest percentage of TD repayment is created by a small subset of issue types [44], and the most frequently introduced TD items are related to low-level coding issues [40]. The most accurate estimations are related to *Code Smells*, while the least accurate to *Bugs* [45], [46].

III. SonarQube and Arcan

In this section, we briefly introduce the two tools used in this work. SonarQube is one of the most used tools for software quality analysis and provides a TD index mainly focused on code level violations, while Arcan provides an Architectural debt index focused on problems/smells. We selected these tools because we have experience using them in previous projects.

A. SonarQube

SonarQube provides a TD index and two remediation estimates[2]. The TD index (also called *squale index* is related to "*the effort (minutes) to fix all Code Smells*". The remediation estimates are related to "*the effort to fix all bug issues*" (reliability remediation effort), and to "*the effort to fix all vulnerability issues*" (security remediation effort). If the analyzed source code violates a coding rule, or if a metric is outside a predefined threshold (also named "gate"), SonarQube generates a "TD issue". The time needed to remove these issues (remediation effort) is used to calculate the remediation cost and technical debt. SonarQube includes reliability, maintainability, and security rules.

SonarQube defines four types of coding rules: *Bugs*, *Code Smells*, *Vulnerabilities*, and *Hot Spots*. *Bugs*, also named Reliability rules, create TD issues that "represent something wrong in the code" and that will soon be reflected in a bug. *Code smells* are considered "maintainability-related issues" in the code that decreases code readability and code modifiability. It is important to note that the term "code smells" adopted in SonarQube does not refer to the commonly known term code smells defined by Fowler et al. [47] but to a different set of rules. *Vulnerabilities* and *Hot Spots* are considered "security-related issues". SonarQube internally uses the SQALE methodology [48] to compute the technical debt ratio and to classify the project to an SQALE rating. The SQUALE rating is based on the "remediation cost". The Remediation cost is defined as the sum of the estimated time to fix all open issues classified as "*Code smells*".

TD ratio is calculated as:

$$TD\ ratio = \frac{TD\ cost}{(Cost\ to\ develop\ 1\ line\ of\ code\ *\ Number\ of\ lines\ of\ code)}$$

[2]https://docs.sonarqube.org/latest/user-guide/metric-definitions/

where the cost to develop a line of code is considered 0.06 days.

Based on the outstanding remediation cost, the project is rated from A to E according to the following rules. For instance, for SonarQube 8.5 the TD ratio $\leq$5%: the rating is A, while the rating for B belongs to 6%$\leq$ TD Ratio$\geq$10%.

B. Arcan

Arcan has been developed for architectural smells (AS) detection and the computation of an Architectural Debt Index (ADI), based on the AS identified in a project [9]. The AS currently detected by Arcan are the following ones:

- *Unstable Dependency (UD)*: describes a component (class or package) that depends on other subsystems that are less stable than the component itself. Detected on packages.
- *Hub-Like Dependency (HL)*: this smell arises when an abstraction has (outgoing and ingoing) dependencies with a large number of other abstractions. It is detected on classes and packages.
- *Cyclic Dependency (CD)*: refers to a component that is involved in a chain of relations that break the desirable acyclic nature of a subsystem's dependency structure.

The ADI computation, integrated into the Arcan tool, takes into account: (i) the *Number* of AS in a project, (ii) the *Severity* of an AS: assuming that some instances of AS are more critical than others, the Index takes into account a Severity measure according to each AS type, and (iii) the *Dependency metrics* of Martin [49] (Instability, Fan In, Fan Out, Efferent, and Afferent Coupling) used for the AS detection. The final *ADI(P)* value quantifies the amount of architectural technical debt present in a project *P*. We also report the *q(ADI(P))*, that is its quantification as a score value in a range from 1 to 5, where 5 is the worst rating a project can have. The value is computed in relation to a reference dataset of over 100 projects from the Qualitas Corpus [50]. If a project is not affected by AS, then the ADI is not computed. All the details about the ADI computation and the reference dataset can be found in a previous work of the authors of the tool [32].

IV. Case Study Design

We designed and conducted a case study [51] to uncover the impact of opportunistic reuse practices in the quality of systems and analyze the technical debt ratios before and after the integration of open-source software (i.e. understood as new functionality) in three existing systems. Therefore, we followed the approach of *exploratory case studies* [52] as a way to explore and identify the overall picture of opportunistic reuse practices. To this aim we raised the following research questions:

RQ$_1$: *Which are the most difficult aspects to integrate new functionality found opportunistically in open source projects?*
Rationale: In this research question we study the effort and the changes required by a developer when has to integrate third-party components, so we can compare the integration effort to develop components from scratch, if necessary.

RQ$_2$: *How code debt is affected after the integration of reusable assets found opportunistically?*
Rationale: Here, we wanted to investigate to what extent opportunistic reuse practices have a significant impact on TD ratios and the number of new smells in the code.

RQ$_3$: *How architectural debt is affected by reusing open-source software?*
Rationale: Like in the previous research question, we are interested to analyze how architectural debt is affected when new components are introduced in existing architectures and what changes induce.

A. Context

In this section, we describe the context of the exploratory study including the case studies we used and the repositories we chose to reuse the components.

1) Case study 1: This case study belongs to a bicycle hiring system (BikeApp) developed at the Telecommunications School from the Technical University of Madrid (UPM) between February-May in 2020. The system promotes sustainable transportation in Madrid renting bikes around the city and using a mobile app. The system is built around an API that uses a Client/Server style and a persistence layer (Java Persistence API - JPA) to access the database. The server side uses J2EE and servlets (i.e. Servlet) supported by Apache Tomcat 9.0. The client-side uses a multi-window approach based on Java Swing. The source code of this small project is only 2.500 SLOC, as we didn't include the third-party libraries required to develop the software. We used the Visual Paradigm tool 16.1[3] to reverse the architecture of the system, shown in the class diagram in Figure 1, which encompasses 52 classes. In the Figure, we can see the main BikeApp class (in brown color), while the UML components (shown in grey color) provide support for the `hsqldb`, `jackess` libraries for accessing the SQL database and a connection to the `EclipseLink` component aimed to link database objects to Java objects handled by the JPA persistence layer. Also, the client uses the `jxmapviewer` graphical library which receives the location of the bikes encoded in JSON format.

2) Case study 2: Our second case study is an open-source system called TEAMMATES [4]. We used version 7.6.0 for our analysis. TEAMMATES is a free online tool for managing confidential peer evaluations for student team projects. The students can evaluate their performance anonymously in team projects and search/view reports of their feedback and evaluations. TEAMMATES was designed to provide powerful peer feedback and peer evaluation mechanism with a very high degree of flexibility. TEAMMATES runs on the top of Google App Engine, using cutting-edge cloud technologies and benefits from the infrastructure that power Google's applications.

3) Case study 3: This case study addresses an IT system Kurki[5], used for managing students participating in courses, their course accomplishment, and the necessary operations to provide grades and other related information. Kurki is a web application, where then frontend relies on HTML, CSS, and JavaScript, and the backend includes Java code for implementing application-specific functions, a web server for processing requests and responses, and an SQL database for storing the information. Designed at the Department of Computer Science, University of Helsinki, Finland, the system has been deployed to use in the 1990s, and many of the design choices still present in the system reflect the state-of-the-art of those days. Over the years, portions of code have disappeared when people working on the system have left the project, resulting in updates in certain parts of the system. Furthermore, numerous developers have participated in the project. Internally, the system includes some legacy code, but things that are related to operating it has been upgraded to today's standards. For instance, to deploy the system, Docker is used, and doing a git push to the master branch and running an associated script are enough to deploy the system.

4) Repositories: To search for reusable components, we selected the following four open-source repositories: *Maven, GitHub, SourceForge, and GitLab*. We based our selection on the following criteria: (i) explanation and documentation of the component including a Javadoc file, (ii) compatible license and version with the target software, (iii) existence of an import file to facilitate the integration process, (iv) access to the source code, (v) functionality required (vi) popularity and (vii) dependencies to other libraries. Some of the aforementioned items are also discussed in [7].

B. Data Collection and Analysis

In this step of the method, we performed the following tasks: (i) we run SonarQube 8.5 for the three projects and we collected the technical debt ratios, the number of smells and issues, (ii) we run the Arcan tool to compute the architectural debt index (ADI) and detect the architectural smells for all the analyzed projects, (iii) we sought in the four repositories for new functionality to extend the three projects, and (iv) once the components found were integrated into the three projects, we run again SonarQube and Arcan to measure the technical debt and other quality ratios provided by the tools. The information about the number of components found and reused is described in Tables II and III. The results of the technical debt rations after reuse are shown in Table V shows. The search and integration efforts were computed manually by two of the authors.

C. Replicability

To allow our study to be replicated, we have published the complete raw data together with the instruction of the assignment and the complete questionnaire in the replication package[6].

[3]https://www.visual-paradigm.com/
[4]https://teammatesv4.appspot.com/web/front/home
[5]https://github.com/UniversityOfHelsinkiCS/kurki

[6] https://github.com/CCS-repository-public/techdebt-2021

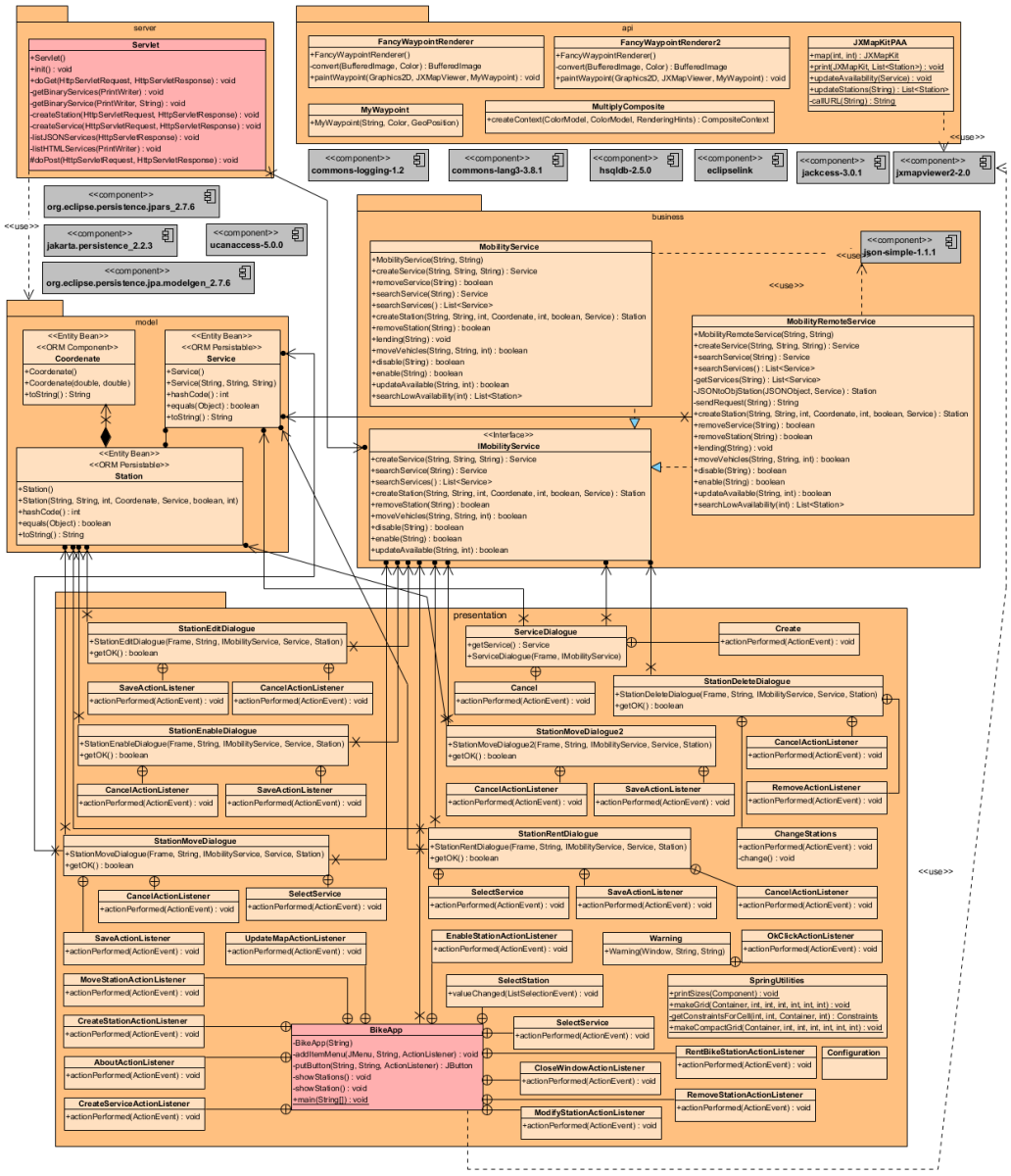

Figure 1. Reversed architecture of the BikeApp software

V. Results

First, we describe the outcome from SonarQube and Arcan tools for the three case studies before extending the functionality with third-party components, such as we describe in Table I. The results we computed include the technical debt ratios (TD ratio), code smells (Smells) and issues (Issues) provided by SonarQube, the number of Architectural Smells (AS) and the Architectural Debt Index (ADI) provided by the Arcan tool and the quantification of ADI (q(ADI)) as a score value in the range from 1 to 5. In order to select and search for reusable assets, (1) we simulated a scenario where the project's team wants to extend the functionalities of that project basing on potential client requirements; (2) we made a list of the new desiderata, and (3) we searched for them in the repositories. The desiderata was brainstormed by three of the co-authors but we didn't follow any specific criterion.

Table I
Results from SonarQube and Arcan before reuse

	SonarQube			Arcan		
Case studies	TD ratio	Smells	Issues	AS	ADI	q(ADI)
BikeApp	5.2	248	253	0	-	-
TEAMMATES	0.6	2087	2905	7	8.0	5
Kurki	1.0	504	536	3	5.0	5

Table II summarizes the number of components we found for each project and the different repositories. For the **BikeApp** case study, we searched for the following functionality: (i) encode the data between the server and the clients, (ii) provide new functionality to geolocate the bikes, and (iii) a calendar.

For the **TEAMMATES** project, we looked for the following functionality: (i) a voting system, and (ii) an event manager. Finally, in the case of **Kurki** as it's similar to TEAMMATES, we decided to include only the event manager functionality so we can compare differences in the TD ratios. For the three projects, we used different keywords in Google as the search string to search for reusable assets in the four repositories.

Table II
NUMBER OF REUSABLE COMPONENTS FOUND IN OPEN-SOURCE REPOSITORIES

Case studies		Maven	GitHub	SourceForge	GitLab
BikeApp	Encode data	115	170	9	13
	Display geolocation	7867	92	5	0
	Calendar	1648	111	8	2
TEAMMATES	Voting system	17	14	92	9
	Event manager	561	80	25	78
Kurki	Event manager	561	80	25	78

In Table III we show the components we found and reused for each project and from which repository we selected those components. The criteria to select a component from a particular repository was based on (i) the description of the functionality of the component, (ii) compatibility of licenses between the software and the reused component (iii) access to the source code, (iv) compatibility of the version of the reused asset with the existing project and dependencies to third-party libraries, (v) facility to import and configure the component, and (vi) existence of a Javadoc file explaining how to use the components found. Other criteria such as popularity or project releases can be also considered. The developer can select one or several of these criteria to find the most suitable asset.

Table III
REUSABLE COMPONENTS SELECTED IN OPEN-SOURCE REPOSITORIES

Case studies	Maven	GitHub	SourceForge	GitLab
BikeApp	Base64	—	Base64	—
	GoogleMaps	GoogleMaps	—	—
	JDatePicker	JDatePicker	—	—
TEAMMATES	Voting-Reward	Voting-Reward, Voting-System	—	—
	—	FullCalendar	—	—
Kurki	—	FullCalendar	—	—

A. Search and Integration effort

In the following, we explain how we integrated the components found in the repositories and the effort taken. We assume we consider the time needed starting from the initial search for each component and project until a component was found and reused, and the time required to integrate the component into the project and check that the application doesn't contain compilation errors and the new functionality is ready to be used. Regarding the search effort, we run Google queries for each new functionality and we used the initial set of results to refine the search in each of the four repositories. If the selected component doesn't serve, we refine the query in each repository or we look for the next popular component. Each repository has different facilities to perform a refined search (e.g.: keywords, categories, or search string).

BikeApp: In this case study we reused 3 components. For the **Base64** component and based on the criteria defined in section 4.1.4, we used the search string *"send java data encoded safety"* in Google and we got around 7 million responses, but the two first pages gave us 21 responses including Base64. We looked for comments about pros and cons and we analyzed the functionality of the component as well as the documentation and other comments. Then, we looked for the selected component in the four repositories and eventually selected the one from SourceForge because it doesn't require dependencies to other libraries. All this effort took around 4.5 hours. The integration was done by importing the asset directly without using Maven. We downloaded those libraries required by the client and the server and we modified the corresponding Classpath file. Then, we modified the Servlet class (on the server-side) to gather the HTTP request needed by the Base64 component and we added two new methods to integrate the reused component. On the client-side, we modified the HTTP request to process the data received. We also created a new class to configure the app and select the type of encoding (i.e. HTML, JSON, Base64). We spent between 1.5 and 1.7 hours in the integration effort.

Regarding the search of the **GMaps** component we looked for an asset that provides the location of the bikes, so the search string we used was *"get gps coordinates in java"*. As a result, we got 1.8 million references. On the two first pages, we found 4 references, and the asset chosen was found in Maven and GitHub repositories. Based on the same criteria as was used in the first case, we selected the component from the Maven repository because it provides compatibility information and which dependencies to other libraries are needed. GitHub does not provide this information in such a clear way. The effort spent in searching the right component, and the integration effort are shown in Table IV.

The integration of this asset required the modification of the class that manages the events to select a hiring bike point and provide the right address encoded in JSON format. We also needed to register into the Google Cloud platform in order to access the Geocode API used to decode the addresses of a GPS location. In addition, we had some connectivity problems during the testing of the reused asset that we solved by installing an additional component required by Google.

Finally, we did the same for a calendar we needed to integrate using Java Swing and the search string we used was *"Java swing calendar"*. We got around 5 million results. From the first two Google pages, we found a reference to the **JDatePicker** component, which was available in Maven and GitHub repositories. We selected the component from GitHub because it doesn't exhibit dependencies to other components and due to the availability of a tutorial and examples of use. We imported the component and we only needed to modify the class that manages the event that activates the calendar. Similar to the other components, the effort spent in the search process and integration as well are shown in Table IV.

TEAMMATES: We did the same for the TEAMMATES projects reusing 2 components. Regarding the **Voting system** component we used the search string *"java student voting systems"* and we got around 9.6 million answers. Screening the first two Google pages, we found two similar components (i.e. voting reward and voting system) in different repositories and reused the *voting system* component according to our criteria but the selection was mainly based on because is an independent Java project that doesn't require dependencies to external projects and also doesn't require changes to be integrated with other software. The second component is an extended calendar including an event manager for appointments. We used the search string *"event manager calendar in java"* and we got 9.8 million results but we reused the **Event manager** component from GitHub because of the examples provided by the third-party developer, documentation of use, as well as references from the other repositories to this component. The results about the components found and reused are shown in Table III, while the effort reusing and integrating both components is available in Table IV.

Kurki: As this is a similar project like TEAMMATES, we decided to integrate the same **FullCalendar** component previously reused so we can compare the trend of the TD ratios of both projects and observe if there are significant architectural differences. In this case the search effort is equal to 0 and the integration effort compared to TEAMMATES just a little bit higher due to differences in the technologies used in both projects. Table IV describes the summary of the search and integration efforts for the different components.

Table IV
SEARCH AND INTEGRATION EFFORTS OF THE REUSED ASSETS (HOURS)

Case studies	assets	Search effort	Integration effort
BikeApp	Base64	4.5	1.7
	GoogleMaps	5	5.5
	JDatePicker	6	1.1
TEAMMATES	VotingSystem	6.5	1.4
	FullCalendar	6	2
Kurki	FullCalendar	0	2.2

Architectural impact after reuse: Finally, we reversed the new architecture of BikeApp including the new components. As we can see in Figure 2, the new classes and components are shown in green color while the entities required by these elements are displayed in light yellow. Some of the new entities in green were created by us to invoke the three new components. We added 13 new classes and components for the new functionality reused, that is an increment of 17% of elements from the original design. It might be possible that reusing different components could have different numbers in terms of new functionality, but what is more important is that the architectural style didn't change after reuse.

B. Debt ratios after reuse

In Table V we describe the results of the technical debt ratios from SonarQube and the ADI index from Arcan after reuse. As we can observe we computed the TD ratios for each project taking into account the SLOC of the new components in a cumulative way (i.e. the TD ratio of the last component of each project includes the previous ones). In the case of the **BikeApp** project, the trend of the TD ratio observed decreased from 4.5 to 1.7. However, the number of smells and issues for SonarQube increased from 248 and 253 (see Table I) to 3551 and 3759 respectively.

From the results of Arcan, we didn't find any architectural debt in the original version of the project, but this debt increased after reuse. Our results show many architectural smells (30 for Base64, 64 for BikeApp-GoogleMaps, and 66 for BikeApp-JDatePicker) with a high q(ADI) value. In particular, Arcan identified the following architectural smells: 1 UD, 43 CD, 3 HL for **Base64**; 18 UD, 43 CD, 3 HL for **GoogleMaps**; 19 UD, 44 CD and 3 HL smells for **JDatePicker**. Concerning the specific types of smell detected, Arcan only identified smells of type Cyclic Dependency. Hence, the values reported in Table V refer only to that type of smell.

Regarding the **TEAMMATES** project, we observed that due to the size of the project (i.e. 128k SLOC) and the small size of the reused components, the TD ratios shown in Table V are the same. Only the number of smells and issues varied. In the case of the voting system, the number of smells grew from 2087 to 2136 and the number of issues increased too from 2905 to 2977. For the event manager component, we got the same number of smells as for the voting system, the increment is insignificant, that is 2979 issues. Regarding Arcan results, the ADI value (both ADI and q(ADI)) did not change since the original project versions. In particular, the ADI value, which equals 12.0 for both extensions, is very close to the ones of BikeApp-GoogleMaps and the BikeApp-JDatePicker. The only change happened in terms of detected smell types: respect the original version, which counted 5 UD and 2 CD, the extended ones show one more UD (for a total of 4) and one less CD (for a total of 3). About **Kurki**, we observed the typical increment in the number of code smells and TD issues compared to the initial project but the TD ratio decreased from 1.0 to 0.6. This is caused because Kurki is not so big a project like TEAMMATES and doubling the number of SLOC after including the new functionality led to a significant reduction of the TD ratio. Regarding Arcan results, as happened for TEAMMATES, the ADI value (both ADI and q(ADI)) did not change from the original project version. Also, the number of smells by type remained the same, i.e., one smell per type.

Table V
TECHNICAL DEBT AND ARCHITECTURAL DEBT RATIOS AFTER REUSE

		SonarQube			Arcan		
Case studies	assets	TD ratio	Smells	Issues	AS	ADI	q(ADI)
BikeApp	Base64	4.5	365	383	30	1.0	3
	GoogleMaps	1.7	3494	3701	64	11.0	5
	JDatePicker	1.7	3551	3759	66	10.0	5
TEAMMATES	VotingSystem	0.6	2136	2977	7	12.0	5
	FullCalendar	0.6	2136	2979	7	12.0	5
Kurki	FullCalendar	0.6	735	678	3	5.0	5

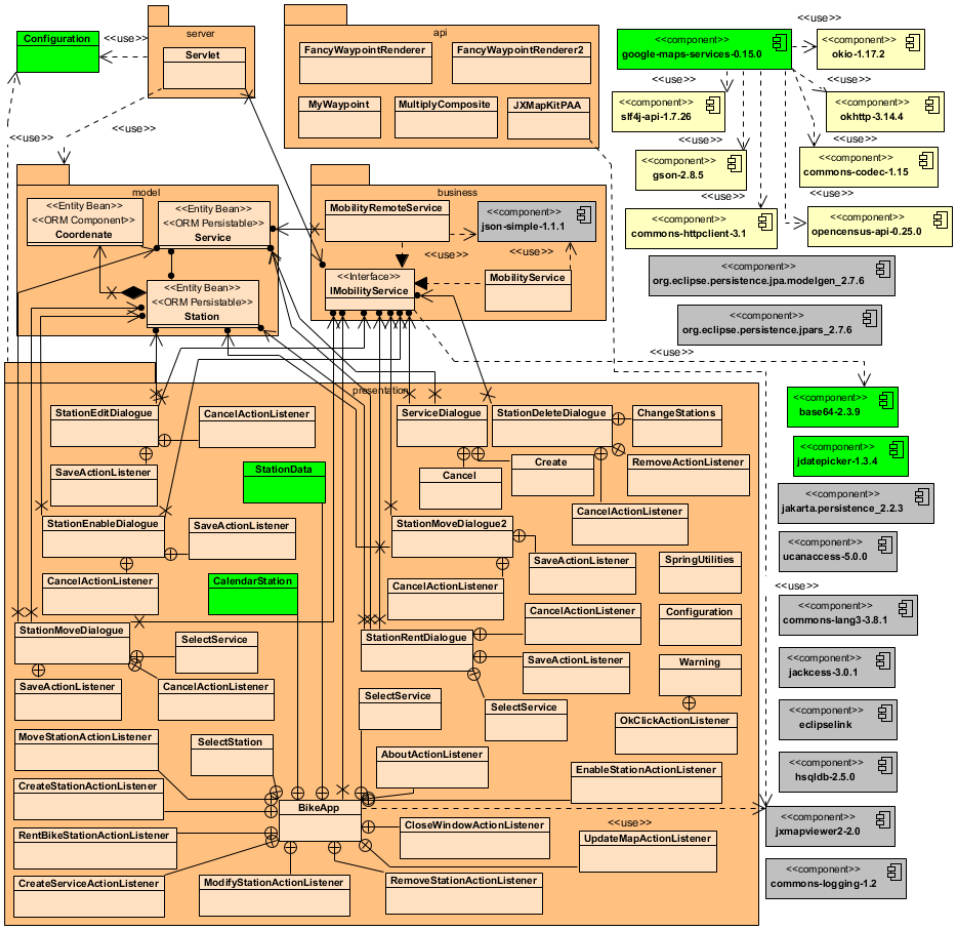

Figure 2. Final version of the reversed architecture of the BikeApp software

VI. Findings

In this section, we describe our findings answering the three research questions.

To answer to **RQ$_1$:** *Which are the most difficult aspects to integrate new functionality found opportunistically in open source projects?*, we can say that the most complex issues we found during the integration of the reused components are the following. First, to identify which elements of the target project needed to be modified are influenced by the architectural style of the project and which are underlying technologies used to implement the architectural style of the project. For instance, for BikeApp and TEAMMATES projects using the model-view-controller (MVC) style we needed to identify the right Web technologies (e.g. BikeApp uses Swing to implement the "view" while TEAMMATES uses Angular).

Second, seeking the right component according to the architectural style and technologies used in the target project, as in other cases the integration of a reused component using different technologies may lead to performing another search. This happened when we integrate the **voting system** component for TEAMMATES. Also, in the case of TEAMMATES and Kurki, we observed certain differences in the integration effort caused by the different technologies used in both projects.

Third, other minor issues like the invocation of the new component or the creation of an instance of the object can be solved by common programming techniques. Independently of some integration problems, we followed the criterion of ease of integration based on a small number of dependencies to third-party libraries and the compatibility of the design pattern of the reused component with the existing application.

From our results and answering to **RQ$_2$:** *How code debt is affected after the integration of reusable assets found opportunistically?*, we observed from Table V that the technical debt ratio decreases if we reuse large-scale components like the **Google Maps**. On the contrary, if we start from a large software project and the sizes of the components are small, the code debt ratio provided by SonarQube remains almost the same. In addition, in the majority of the cases, the number of smells and issues increase, but we found one case (i.e. the voting system component) where the number of issues decreased a bit. Another aspect not covered in this study but worthy to be investigated is the quality and severity of the new smells and issues and not counting only the number.

About **RQ$_3$:** *How architectural debt is affected by reusing open-source software?*, our results show that reuse has an impact in terms of architectural smells and consequently of

architectural debt. In general, from what we observed in Table V, architectural debt increases after reuse, and in particular when reusing large-scale components (**Google Maps** and **JDatePicker**). In the case of project TEAMMATES, where the number of AS does not change after reuse, the debt increases too, meaning that the AS worsen in terms of *severity* (see Section III-B), e.g., they grow in size and affect more components (classes and packages). The only project which was not affected by reuse is Kurki, whose number of architecture smells (AS) and ADI values remained the same. Nevertheless, one important aspect once open-source software is reuse refers to those components that can be integrated without performing significant architectural changes. In some cases, we needed additional search effort to find suitable components aligned by the technologies supporting the architecture of the project.

VII. Threats to Validity

Some factors might have influenced the results reported in our study. We discuss the main threats to validity and how we mitigated them according to Yin's guidelines [52].

Construct Validity. We adopted the default set of collected measures considered by the SonarQube model since practitioners are reluctant to customize the built-in quality gate and mostly rely on the standard set of rules [53]. Also, we have tried as well as possible to replicate the conditions adopted by practitioners that use this tool, although we are aware that the detection accuracy of some rules may not be precise.

Internal Validity. SonarQube detected duplication of the same issue, reporting the issue violated in the same class and in the same position but with different resolution times. We are aware of this fact, but we did not remove such issues from the analysis since we wanted to report the results without modifying the output provided by SonarQube and introducing other biases in the study.

External Validity. We analyzed three case systems trying to select different projects with different characteristics. However, we are aware that other projects might present slightly different results. We have considered for architectural debt detection only the AS detected by the Arcan tool. We could have different results by considering other AS, but these smells based on dependency issues are certainly particularly critical for a project. We can mitigate this threat using in the future another tool to validate our initial results detecting architecture smells.

Conclusion Validity. We can rely on the two tools we selected. SonarQube is one of the most popular static analysis tools largely adopted both in academia [54], [55] and in industry [53]. Validation of Arcan results has been performed on ten open-source projects [56], on two industrial projects, with a high precision value of 100% in the results and 63% of recall [9] and through the feedback provided by practitioners working on four industrial projects [29].

VIII. Conclusion and Future Work

To the best of our knowledge, this is the first paper that examines the impact of reusing software components opportunistically in different repositories to TD. Our main findings, investigated to answer the three research questions, show that for larger projects the TD ratio provided by SonarQube remains stable or decreases, but in most cases, the number of code smells and TD issues increase. The same happens when we add a large component to a small project. We also evaluated the projects' architectural debt and the number of architectural smells before and after reuse. Similar to SonarQube, Arcan detected more architectural smells after reuse, which resulted in increasing architectural debt. One interesting outcome is that in the most affected project (BikeApp), the most common type of smell is Cyclic Dependency, suggesting that developers should particularly pay attention to this specific smell while reusing software. This work, being a case study, does not provide conclusive evidence of the effect of opportunistic reuse on TD. Therefore, further research is needed on the topic, to better understand the right approach to measure TD in relation to opportunistic code reuse.

In future work, we plan to evaluate these initial trends in more projects and repositories and compare them to programs implemented with some other programming languages like Python. We also plan to investigate how to reduce the initial search effort when the first component found doesn't satisfactorily serve the project's needs.

References

[1] P. Kruchten, R. Nord, and I. Ozkaya, *Managing Technical Debt: Reducing Friction in Software Development*. Addison-Wesley Professional, 2019.

[2] V. Lenarduzzi, T. Besker, D. Taibi, A. Martini, and F. Arcelli Fontana, "A systematic literature review on technical debt prioritization: Strategies, processes, factors, and tools," *Journal of Systems and Software*, vol. 171, 2021.

[3] R. Verdecchia, P. Kruchten, and P. Lago, "Architectural technical debt: A grounded theory," in *14th European Conference on Software Architecture ECSA 2020*, 2020, pp. 202–219.

[4] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice," *IEEE Softw.*, vol. 29, no. 6, pp. 18–21, 2012.

[5] Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management," *J. Syst. Softw.*, vol. 101, pp. 193–220, 2015.

[6] M. Griss, "Systematic software reuse – it isn't what it used to be!" Keynote presented at the the 14th International Conference on Software Reuse, Miami, US, 2015. [Online]. Available: http://icsr2015.ipd.kit.edu/keynotes/index.html

[7] D. Spinellis, "How to select open source components," *Computer*, vol. 52, no. 12, pp. 103–106, 2019.

[8] I. Zozas, A. Ampatzoglou, S. Bibi, A. Chatzigeorgiou, P. Avgeriou, and I. Stamelos, "REI: an integrated measure for software reusability," *J. Softw. Evol. Process.*, vol. 31, no. 8, 2019.

[9] F. Arcelli Fontana, I. Pigazzini, R. Roveda, D. A. Tamburri, M. Zanoni, and E. D. Nitto, "Arcan: A tool for architectural smells detection," in *Int'l Conf. Software Architecture (ICSA)*, 2017, pp. 282–285.

[10] M. Morisio, M. Ezran, and C. Tully, "Success and failure factors in software reuse," *IEEE Transactions on software engineering*, vol. 28, no. 4, pp. 340–357, 2002.

[11] M. L. Griss, "Software reuse architecture, process, and organization for business success," in *Proceedings of the Eighth Israeli Conference on Computer Systems and Software Engineering*. IEEE, 1997, pp. 86–89.

[12] B. Jalender, A. Govardhan, and P. Premchand, "A pragmatic approach to software reuse." *Journal of Theoretical & Applied Information Technology*, vol. 14, 2010.

[13] W. Tracz, *Confessions of a Used Program Salesman*. Addison-Wesley, 1995.

[14] C. W. Krueger, "Software Reuse," *ACM Computing Surveys*, vol. 24, no. 2, pp. 131–183, 1992.

[15] T. Mikkonen and A. Taivalsaari, "Software reuse in the era of opportunistic design," *IEEE Software*, vol. 36, no. 3, pp. 105–111, 2019.

[16] E. Petrinja, R. Nambakam, and A. Sillitti, "Introducing the opensource maturity model," in *2009 ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development*. IEEE, 2009, pp. 37–41.

[17] B. Golden, "Making open source ready for the enterprise: The open source maturity model," *Open Source Business Resource*, 2008.

[18] A. Raza, L. F. Capretz, and F. Ahmed, "An open source usability maturity model (os-umm)," *Computers in Human Behavior*, vol. 28, no. 4, pp. 1109–1121, 2012.

[19] T. Kilamo, T. Aaltonen, I. Hammouda, T. J. Heinimäki, and T. Mikkonen, "Evaluating the readiness of proprietary software for open source development," in *IFIP International Conference on Open Source Systems*. Springer, 2010, pp. 143–155.

[20] A. Zahoor, K. Mehboob, S. Natha *et al.*, "Comparison of open source maturity models," *Procedia computer science*, vol. 111, pp. 348–354, 2017.

[21] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed, "Enhancing clone-and-own with systematic reuse for developing software variants," in *30th IEEE International Conference on Software Maintenance and Evolution,*, 2014, pp. 391–400.

[22] R. Capilla, B. Gallina, C. Cetina Englada, and J. Favaro, "Opportunities for software reuse in an uncertain world: From past to emerging trends," *Journal of Software: Evolution and Process*, vol. 31, no. 8, 2019.

[23] N. Ali, D. Horn, and J.-E. Hong, "A hybrid devops process supporting software reuse: A pilot project," *Journal of Software Evolution and Process*, pp. 1–23, 2020.

[24] N. Mäkitalo, A. Taivalsaari, A. Kiviluoto, T. Mikkonen, and R. Capilla, "On opportunistic software reuse," *Computing*, vol. 102, no. 11, pp. 2385–2408, 2020.

[25] T. Sharma, P. Singh, and D. Spinellis, "An empirical investigation on the relationship between design and architecture smells," *Empir. Softw. Eng.*, vol. 25, no. 5, pp. 4020–4068, 2020.

[26] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Identifying architectural bad smells," in *13th European Conference on Software Maintenance and Reengineering, CSMR*, 2009, pp. 255–258.

[27] D. M. Le, C. Carrillo, R. Capilla, and N. Medvidovic, "Relating architectural decay and sustainability of software systems," in *13th Working Conference on Software Architecture, WICSA*, 2016, pp. 178–181.

[28] R. Verdecchia, I. Malavolta, and P. Lago, "Architectural technical debt identification: the research landscape," in *International Conference on Technical Debt, (TechDebt'18)*, 2018, pp. 11–20.

[29] A. Martini, F. Arcelli Fontana, A. Biaggi, and R. Roveda, "Identifying and prioritizing architectural debt through architectural smells: a case study in a large software company," in *European Conf. on Software Architecture (ECSA'18)*, 2018.

[30] M. S. Laser, N. Medvidovic, D. M. Le, and J. Garcia, "ARCADE: an extensible workbench for architecture recovery, change, and decay evaluation," in *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1546–1550.

[31] G. Suryanarayana, G. Samarthyam, and T. Sharma, *Refactoring for Software Design Smells: Managing Technical Debt*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2014.

[32] R. Roveda, F. Arcelli Fontana, I. Pigazzini, and M. Zanoni, "Towards an architectural debt index," in *Euromicro Conference on Software Engineering and Advanced Applications (SEAA'18)*. IEEE, 2018.

[33] F. Arcelli Fontana, R. Roveda, and M. Zanoni, "Technical debt indexes provided by tools: A preliminary discussion," in *2016 IEEE 8th Inter. Work. on Managing Technical Debt (MTD)*, Oct 2016, pp. 28–31.

[34] P. Avgeriou, D. Taibi, A. Ampatzoglou, F. Arcelli Fontana, T. Besker, A. Chatzigeorgiou, V. Lenarduzzi, A. Martini, N. Moschou, I. Pigazzini, N. Saarimäki, D. Sas, S. Soares de Toledo, and A. Tsintzira, "An overview and comparison of technical debt measurement tools," *IEEE Software*, 2021.

[35] W. Wu, Y. Cai, R. Kazman, R. Mo, Z. Liu, R. Chen, Y. Ge, W. Liu, and J. Zhang, "Software architecture measurement - experiences from a multinational company," in *12th ECSA, 2018*, pp. 303–319.

[36] R. Verdecchia, P. Lago, I. Malavolta, and I. Ozkaya, "Atdx: Building an architectural technical debt index," in *Evaluation of Novel Approaches to Software Engineering (ENASE)*, 2020.

[37] C. Seaman and Y. Guo, "Measuring and monitoring technical debt," *Advances in Computers*, vol. 82, pp. 25 – 46, 2011.

[38] V. Lenarduzzi, A. Martini, D. Taibi, and D. A. Tamburri, "Towards surgically-precise technical debt estimation: Early results and research roadmap," in *International Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE'19)*, 2019, pp. 37–42.

[39] I. Tollin, F. Arcelli Fontana, M. Zanoni, and R. Roveda, "Change prediction through coding rules violations," in *21st International Conference on Evaluation and Assessment in Software Engineering (EASE'17)*, 2017, pp. 61–64.

[40] N. Saarimäki, V. Lenarduzzi, and D. Taibi, "On the diffuseness of code technical debt in java projects of the apache ecosystem," in *Second International Conference on Technical Debt (TechDebt '19)*, 2019, pp. 98–107.

[41] V. Lenarduzzi, N. Saarimäki, and D. Taibi, "Some sonarqube issues have a significant but small effect on faults and changes. a large-scale empirical study," *Journal of Systems and Software*, vol. 170, p. 110750, 2020.

[42] D. Falessi, B. Russo, and K. Mullen, "What if i had no smells?" in *International Symposium on Empirical Software Engineering and Measurement (ESEM2017)*, 2017.

[43] V. Lenarduzzi, F. Lomio, H. Huttunen, and D. Taibi, "Are sonarqube rules inducing bugs?" in *27th International Conference on Software Analysis, Evolution and Reengineering (SANER2020)*, 2020, pp. 501–511.

[44] 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?" 2018.

[45] V. L. N. Saarimaki, M.T. Baldassarre and S. Romano, "On the accuracy of sonarqube technical debt remediation time," in *Euromicro Conference on Software Engineering and Advanced Applications (SEAA'19)*, 2019.

[46] M. T. Baldassarre, V. Lenarduzzi, S. Romano, and N. Saarimäki, "On the diffuseness of technical debt items and accuracy of remediation time when using sonarqube," *Information and Software Technology*, vol. 128, 2020.

[47] M. Fowler and K. Beck, "Refactoring: Improving the design of existing code," *Addison-Wesley Longman Publishing Co., Inc.*, 1999.

[48] K. Mordal-Manet, F. Balmas, S. Denier, S. Ducasse, H. Wertz, J. Laval, F. Bellingard, and P. Vaillergues, "The squale model — a practice-based industrial quality model," in *2009 IEEE International Conference on Software Maintenance*, 2009, pp. 531–534.

[49] R. C. Martin, "Object oriented design quality metrics: An analysis of dependencies," *ROAD*, vol. 2, no. 3, 1995.

[50] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "The qualitas corpus: A curated collection of java code for empirical studies," in *2010 Asia Pacific Software Engineering Conference*. IEEE, 2010, pp. 336–345.

[51] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, no. 2, pp. 131–164, 2009.

[52] R. K. Yin, *Case Study Research Design and Methods (5th ed.)*. Sage, 2014.

[53] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, H. C. Gall, and A. Zaidman, "How developers engage with static analysis tools in different contexts," *Empirical Software Engineering*, 2019.

[54] V. Lenarduzzi, A. Sillitti, and D. Taibi., "Analyzing forty years of software maintenance models," in *39th International Conference on Software Engineering Companion*, ser. ICSE-C '17, 2017, pp. 146–148.

[55] V. Lenarduzzi, A. Sillitti, and D. Taibi, "A survey on code analysis tools for software maintenance prediction," in *6th International Conference in Software Engineering for Defence Applications*. Springer International Publishing, 2020, pp. 165–175.

[56] F. Arcelli Fontana, I. Pigazzini, R. Roveda, and M. Zanoni, "Automatic detection of instability architectural smells," in *Proc. of the 32nd Intern. Conf. on Software Maintenance and Evolution (ICSME 2016)*. Raleigh, North Carolina, USA: IEEE, Oct. 2016, eRA Track.