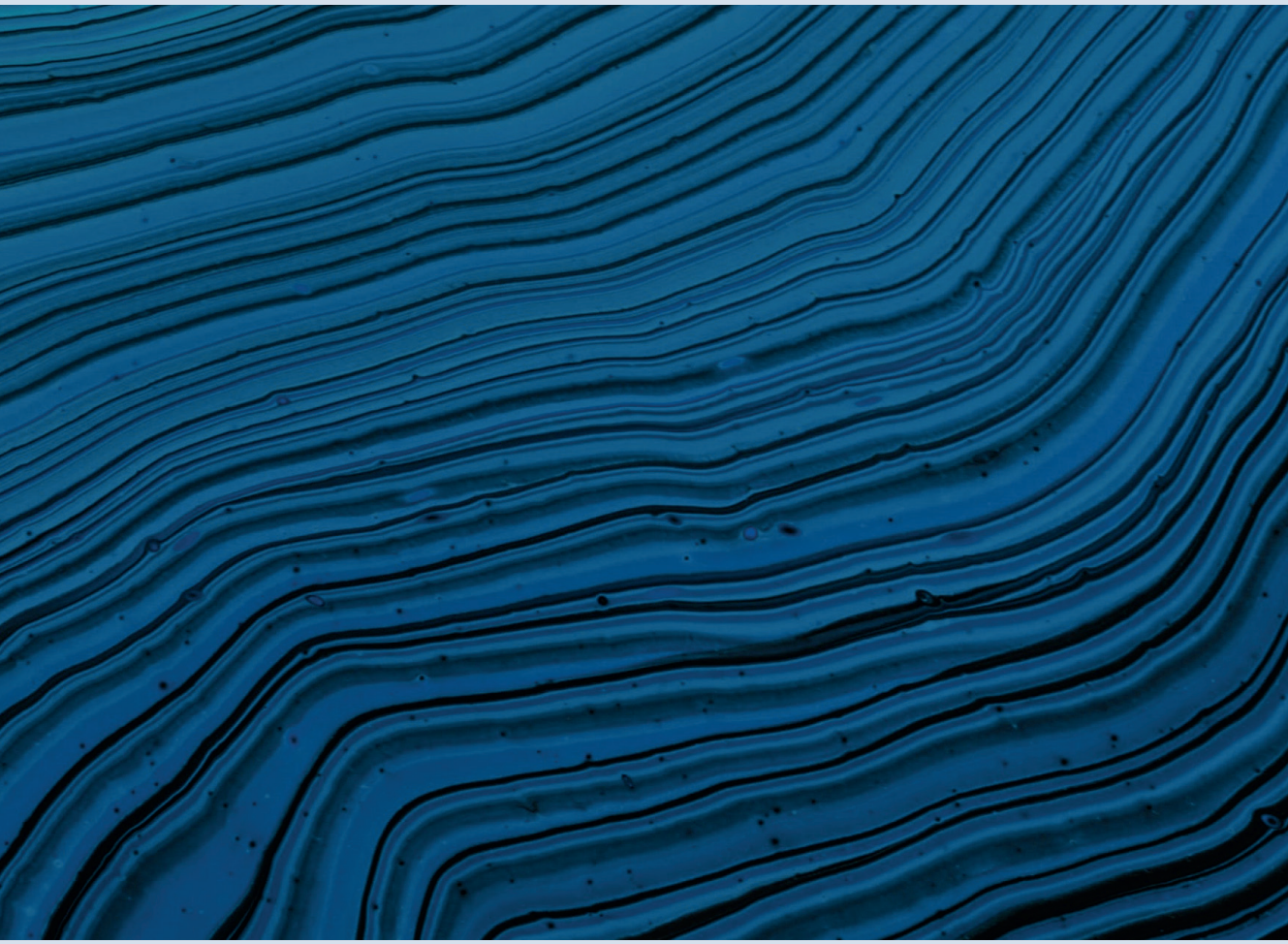
Architectural Technical Debt: Identification and Management

Roberto Verdecchia



GRAN SASSO SCIENCE INSTITUTE
SCHOOL OF ADVANCED STUDIES
Scuola Universitaria Superiore

VU UNIVERSITY AMSTERDAM

SOFTWARE AND SUSTAINABILITY
S2 RESEARCH GROUP
VRIJE UNIVERSITEIT AMSTERDAM

Ph.D. in Computer Science - Doctoral Thesis

Architectural Technical Debt: Identification and Management

Roberto Verdecchia

Promotors

Prof. Dr. Patricia Lago

Prof. Dr. Rocco De Nicola

Co-Promotors

Dr. Ivano Malavolta

Dr. Catia Trubiani

Doctoral Committee

Prof. Dr. Jan Bosch

Dr. Antonio Martini

Dr. Damian A. Tamburri

Dr. Eoin Woods

Prof. Dr. Henri Bal

A thesis submitted in fulfillment of the requirements

for the degree of Doctor of Philosophy at the

Gran Sasso Science Institute

Viale Francesco Crispi 7, 67100, L'Aquila, Italy

and the

Vrije Universiteit Amsterdam

De Boelelaan 1105, 1081HV Amsterdam, The Netherlands

September 10, 2021

SIKS Dissertation Series No. 2021-19
The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.
Promotiecommissie:
Prof. Dr. Jan Bosch (Chalmers University of Technology, Gothenburg, Sweden)
Dr. Antonio Martini (University of Oslo, Oslo, Norway)
Dr. Damian A. Tamburri (Eindhoven University of Technology, Eindhoven, The Netherlands, and Jheronimus Academy of Data Science, s'Hertogenbosch , The Netherlands)
Dr. Eoin Woods (Endava, London, United Kingdom)
Prof. Dr. Henri Bal (Vrije Universiteit Amsterdam, The Netherlands)

ISBN 978-94-6423-368-1

Cover design by Silvia Agozzino, MUTTNIK
Cover image credits to Pawel Czerwinski
Published by ProefschriftMaken
Typeset in LaTeX by Roberto Verdecchia

VRIJE UNIVERSITEIT

Architectural Technical Debt: Identification and Management

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van Doctor of Philosophy
aan de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. V. Subramaniam,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de Faculteit der Bètawetenschappen
op vrijdag 10 september om 9.45 uur
in de online bijeenkomst van de universiteit,
De Boelelaan 1105

door

Roberto Verdecchia

geboren te Bagno a Ripoli, Italië

promotoren: prof.dr. P. Lago
prof.dr. R. De Nicola

copromotoren: dr. I. Malavolta
dr. C. Trubiani

Declaration

I, Roberto Verdecchia, declare that this thesis titled “Architectural Technical Debt: Identification and Management” and the work presented in it are my own.
I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this Universities.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at these Universities or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

10 September 2021

"We build our computers the way
we build our cities: over time,
without a plan, on top of ruins."
— Ellen Ullman

Abstract

Architectural technical debt (ATD) in a software-intensive system is the sum of all design choices that may have been suitable or even optimal at the time they were made, but which today are significantly impending progress: structure, framework, technology, languages, etc. Unlike code-level technical debt which can be readily detected by static analysers, and can often be refactored with minimal or only incremental efforts, architectural debt is hard to detect, and its remediation rather wide-ranging, daunting, and often avoided. The objective of this thesis is to develop a better understanding of architectural technical debt, and determine what strategies can be used to identify and manage it. In order to do so, we adopt a wide range of research techniques, including literature reviews, case studies, interviews with practitioners, and grounded theory. The result of our investigation, deeply grounded in empirical data, advances the field not only by providing novel insights into ATD related phenomena, but also by presenting approaches to pro-actively identify ATD instances, leading to its eventual management and resolution.

Sommario

Architectural technical debt (ATD) in un sistema ad uso intensivo di software è la somma di tutte quelle scelte progettuali che risultavano adeguate o addirittura ottimali al momento in cui sono state fatte, ma che oggi rallentano significativamente il progresso del sistema: struttura, *framework*, tecnologia, linguaggi, etc. A differenza del *technical debt* a livello di codice, che può essere facilmente rilevato tramite l'analisi statica del codice e spesso può essere risolto *via refactoring* con sforzi minimi o solamente incrementali, l'*architectural technical debt* è difficile da rilevare, e la sua riparazione è ampia, scoraggiante, e spesso evitata. L'obbiettivo di questa tesi è sviluppare una migliore comprensione dell'*architectural technical debt*, e determinare quali strategie possono essere utilizzate per identificarlo e gestirlo. Per raggiungere il nostro obbiettivo, adottiamo un'ampia gamma di tecniche di ricerca, tra cui revisioni della letteratura, casi di studio, interviste con professionisti, e *grounded theory*. Il risultato della nostra indagine, profondamente fondata su dati empirici, avanza il campo di ricerca non solo fornendo nuova conoscenza sui fenomeni correlati all'ATD, ma anche presentando approcci per identificare in modo proattivo le istanze di ATD, portando alla sua eventuale gestione e risoluzione.

Samenvatting

Architectural technical debt (ATD) in een software-intensief systeem is de som van alle ontwerpkeuzes die mogelijkerwijs geschikt of zelfs optimaal waren op het moment dat ze werden gemaakt, maar de geschiktheid van die keuzes is zeer onderhevig aan de voortgang binnen het domein: structuur, *frameworks*, technologie, talen, enz. *Technical debt* op het niveau van programmeercode kan gemakkelijk worden gedetecteerd door analyses gebaseerd op statistische methoden. Deze code kan vaak worden aangepast met minimale inspanningen. ATD daarentegen is moeilijk te detecteren, en de aanpassing ervan is erg arbeidsintensief en wordt daardoor vaak vermeden. Het doel van dit proefschrift is om een beter begrip te krijgen van ATD, en te bepalen welke strategieën kunnen worden gebruikt om deze te identificeren en te beheren. Om dit te doen, passen we een breed scala aan onderzoekstechnieken toe, waaronder literatuuroverzichten, case studies, interviews met praktijkmensen en *grounded theory*. Het resultaat van ons onderzoek, dat diepgeworteld is in empirische data, draagt niet alleen bij aan vooruitgang in het veld door nieuwe inzichten te verschaffen in ATD-gerelateerde fenomenen, maar ook door benaderingen te presenteren om ATD-instanties proactief te identificeren, wat leidt tot het uiteindelijke beheer en de oplossing ervan.

Contents

1 Introduction

It always takes longer than you expect, even when you take into account Hofstadter's Law.

Douglas Hofstadter, *"Gödel, Escher, Bach: An Eternal Golden Braid"*

This chapter is based on R. Verdecchia, *Identifying Architectural Technical Debt: Moving Forward*, IEEE International Conference On Software Architecture (ICSA), 2018 [1].

1.1 Backround

In software development processes, a high number of heterogeneous (and from time to time even conflicting) goals have to be considered. Fulfillment of functional requirements, adherence to quality standards, time to market (TTM) and budget management are among the constraints that steer the administration of development processes. This leads to the establishment of a set of tradeoffs that have to be considered in order to deliver software products by meeting all the prefixed goals. To have a more clear picture, think of the development of a software product that has to be delivered by a hard deadline. The development team wants to ensure the highest quality of the product possible, but the timeframe available is too short. Developers hence decide to lower a bit their quality standards in order to deliver in time a final product that fully adheres to the contracted specifications, even if it is of a slightly lower quality of the one envisioned. In particular, under pressure of constraints such as budget or TTM developers tend to introduce in code bases *workarounds*, *shortcuts*, or "*hacks*" in order to implement as fast as possible a functional requirement of the software product. While such implementation solutions can support developers in achieving their short-term goals, the presence of such artifacts tends to incrementally deteriorate the quality of code bases, to the point to which a refactoring process is required.

In 1992 Cunningham coined the term *technical debt* (TD). Such metaphor, relating a software engineering phenomena to economical debt, is used to describe a design or construction heuristics chosen as expedients in the short short term, but which creates a technical context that increases complexity and cost in the long term [2]. By quoting Cunningham's work:

> *"Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. Objects make the cost of this transaction tolerable. The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object-oriented or otherwise"* [3].

In software intensive systems technical debt is hence a metaphor utilized to describe sub-optimal software design solutions (usually adopted to speed up the development process) which increase the effort required to modify the software-intensive system in the long run. In particular, internal software quality attributes such as

maintainability and evolvability seem to be drastically effected by technical debt related issues. Instances of technical debt in a software-intensive system are usually referred to as *TD Items* (TDI). The characterization of such elements entails different concepts. Starting from the economical viewpoint, each TDI is associated to two costs: the *principal* and the *interest amount*. The principal cost of a TDI represents the investment required, in monetary or temporal terms, to eliminate the item. The interest amount instead represents the penalty incurred if a TDI is not solved in the present [4]. This second definition of TD cost entails the growing cost associated to an unmodified TDI during the evolution of a software-intensive system. In general terms, similarly to software maintainability, if no effort is put in refactoring processes, the overall TD of the systems tends to grow in a monotonic fashion [5].

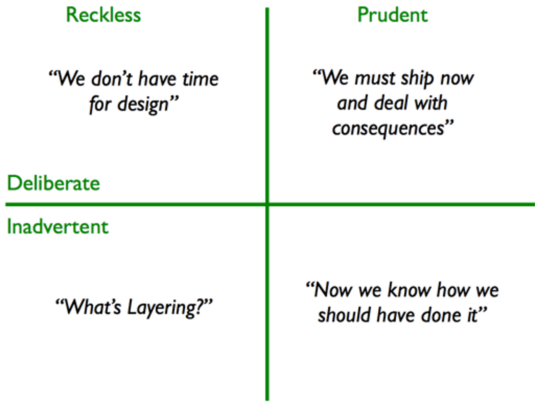


Figure 1.1 – Characterization of TD [6]

Flower [6] further characterized TDIs by introducing an additional characteristic connected to the knowledge that developers have w.r.t. the items. In particular, a design shortcut can be taken *deliberately*, i.e. with the intention of introducing a new TDI, or *inadvertently*, i.e. without the knowledge of the developer of the insertion of a new TDI. Another important distinction that Flower makes is between *reckless* and *prudent* TDIs. A reckless TDI is characterized by a prohibitive interest payment or a long time span that is required to pay down the principal. In contrast, a prudent TDI is characterized by a very low interest rate, that in many cases is not even worth to be repaid, e.g., if the item appears in a portion of the code base that is only seldom modified. Such attributes of TDIs makes it possible to classify them in four distinct

categories, associated to the four quadrants depicted in Figure 1.1.

McConnell in his taxonomy [7], considers instead some different aspects of TDI characterization, namely *debt repayment time* and *TD size*. More in detail, *debt repayment time* can be of two different types: *short-term* and *long-term*. *Short-term debt* refers to debt that has to be paid off frequently as a reactive tactical decision, e.g., a late-stage measure to timely respect a deadline release. *Long-term debt* instead encompasses strategic decisions, such as the (ab)use of a technology, leading to an anchoring bias that negatively influences the evolution of a software-centric system in terms of internal and external quality. *TD size* instead considers the relative size of TDI items. In particular, some TDIs can be taken on in large chuncks, e.g. by deciding to ship a software product with a missing functionality that will be patched later. Other TDIs can instead be accumulated through time with only a small gradual increment of the overall TD. Such TDIs are usually code related, and reperesent small shortcuts, suc as generic variable names, sparse comments, not following code conventions etc.

Kruchten et al. [8] further characterize the TD ecosystem by providing an additional organization of the landscape, as depicted in Figure 1.2. In particular, the proposed landscape organization outlines the possible software improvements of a system from a given state. In particular, a first distinction is made between visible elements (e.g. additional functionality and low external quality) and invisible ones (e.g. documentation debt and code complexity). In addition, on the left hand side of Figure 1.2 we have clustered TDIs related to software evolution and related challenges. On the right hand side instead, we have issues related to the quality of software-centric systems.

In general, the TD research field is characterized by a set of heterogeneous research activities [9], namely:

1. *Identification*: This activity concerns the theoretical and practical aspects of identifying TDIs through the inspection of software related artifacts (e.g. code bases, commit logs, and documentation);
2. *Measurement*: This activity, tightly coupled with the first one, entails the conceivement and evaluation of metrics to measure the impact or overall presence of TDIs in a software-intensive system;
3. *Management*: Once TDIs are identified and quantified in terms of number and impact, prioritization and prevention techniques can be deployed in order to

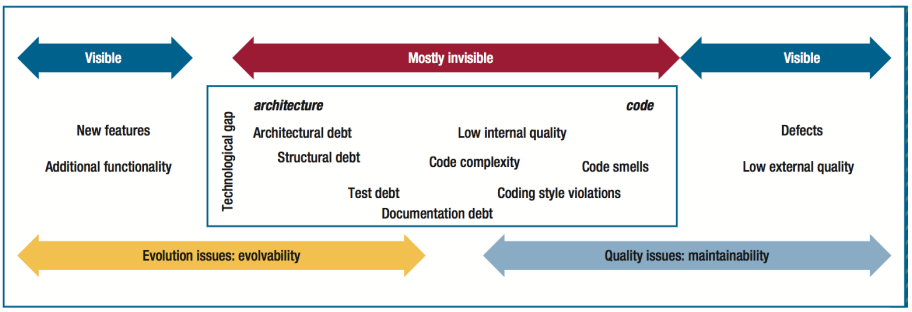


Figure 1.2 – The technical debt landscape. On the left, evolution or its challenges; on the right, quality issues, both internal and external. [8]

keep under control the overall TD of a software-intensive system. This activity relates to the management of known TDIs.

4. *Evolution monitoring*: By considering the evolution of software systems through time, assertions can be made on the future behavior of TDIs and their triggering causes. This activity concerns getting a deeper understanding of TD by considering historical data of software-intensive systems.

5. *Representation\documentation*: In order to keep track of the TD of a software-intensive system it is important to represent and document the identified (and eventually measured) TDIs. This activity concerns the means through which TD can be represented and documented uniformly.

6. *Perception of TD*: This activity concerns the communication of TD related topics to the stakeholders. While the negative economical impact of TD on software systems is well known [5], effort has to be spent in order to convey information regarding the relevance of TD to stakeholders unfamiliar with the topic.

As will be further discussed in the following section, from an inspection of the related literature, not all activities of the TD research result to be studied with the same intensity. In particular, several gaps of research can be identified by considering the secondary studies concentrating on TD [10, 11, 9, 4, 5, 12]. From such studies several TD related activities result to be lacking the required depth and in some cases remain

unexplored. This trend is confirmed by experts of the field, and is generally accepted in the most prominent TD workshops and symposiums [13, 14][1].

1.2 Types of Technical Debt

In addition to the research activities carried out on TD, studies related to this area are also characterized by the different types of TD considered. Each of this TD types has its own level of abstraction and artifact(s) as input. Several coarse-grained types of TD can be identified in the literature, as reported in the second studies available on TD [11, 9, 4]. Some among the most prominent TD types that can be found in the literature are reported in Table 1.1.

The classification of TD into types reported in Table 1.1 provides an eagle-eye overview of the most common TDIs considered in the literature. Such brief classification eases the reasoning on the multifaceted aspects that are entailed by the TD metaphor. In the following sections such classification will be adopted to report the most prominent research trends according to the TD body of literature.

[1] Supported by the growing interest of researchers and practitioners, the first conference focusing on TD was recently announced, and will be co-located with the 40th International Conference on Software Engineering (ICSE) in 2018 [15]

Table 1.1 – Overview of prominent TD types documented in the literature

TD Type	Description
Requirements TD	Inconsistencies that can be found between the actual implementation of a software-intensive system and its optimal requirements [16].
Architectural TD	Sub-optimal architectural design decisions that result to negatively affect the non-functional requirements of a software-intensive system [17, 18, 19, 20, 21, 22, 23, 24].
Design TD	"Shortcuts" taken at the detailed implementation design level resulting in an increase of overall TD of a software-intensive system [12, 25].
Code TD	Poorly written code violating prominent coding practices. Examples of code-related antipatterns are documented by Brown et al. [26].
Test TD	TD resulting from poor testing practices, e.g., lack of testing [27, 28].
Build TD	Difficulties encountered during the building phase of a software system, resulting in overly complex building procedures or problems encountered at build time [29, 30].
Documentation TD	Lack of documentation of a software system. These TD items can be present at different levels of abstraction, from unorganized architectural documentation to lack of code comments [31, 32].
Infrastructure TD	Sub-optimal design solutions taken at infrastructural level, such as technology choice and development-related processes [33].
Defect TD	TD occurring due to the presence of bugs in software systems, e.g. the presence of uncovered or defferred bugs in a software system [34].
Service TD	Sub-optimal design choices related to Service Oriented Architectures, e.g., incorrect service selection, or ill-fitted service composition. [35].
Versioning TD	Incorrect management of source code versioning, e.g. unnecessary forks in code repositories.
People TD	TD deriving from sub-poptimal socio-technical decisions, e.g., agile methods selection or outsourcing [36, 37].

1.3 Overview of Technical Debt Research Trends

From the eagle-eye overview of the types of TD and by considering the data reported in the secondary studies, we can draw some summary conclusions on the most noticeable trends of the TD research field. In particular, a first consideration can be made on the research areas that result to be more active, and which instead result to be only marginally investigated. From the results reported by Li et al. [20] we can notice that *Code TD* results to be the most studied subject, appearing in approximately 40% of the total number of researches they took into account (38 out of 94). As further conjectured by Alves et al. [4] this trend can be attributed to the ease of *Code TD* assessment, for which many tool supported metrics are already established. Another possible explanation of such trend relies on the growing body of knowledge of TD. In fact, while *Code TD* has been studied for many years [38], other types of TD result to date to be fairly young research fields, that were only studied in the most recent years [4]. Nevertheless, as pointed out by all the secondary studies on TD present in the literature [10, 11, 9, 4, 5] TD related analysis cannot be carried out at code level alone, as some TDIs can be identified exclusively by considering higher levels of abstraction.

Based on the data of the secondary studies, the second most studied type of TD results to be the *Architectural TD* (ATD). From the analysis of some of the most prominent studies related to this topic [18, 19, 20, 21, 11, 22, 23, 39, 17, 24] a common trend can be noticed: how to identify, measure and manage Architectural TDIs (ATDI) is still an open question. While the remainder of this section is dedicated to the presentation of TD related research trends, further considerations on ATD research activities are reported in Section 1.5. Following ATD, the remaining of the research activities focus mostly on *Desing TD* and *Test TD*, which according to Li et al. correspond respectively to the 26% (24 out of 94 studies) and 24% (23 studies) of the totality of the literature selected for their mapping study. These trends confirm again the current focus of TD related research, which entails mostly activities considering code bases as the main artifact to gather empirical data. The remaining of the TD types reported in Section 1.2 result generally to be less studied, and are in most of the cases reported to appear in less than the 10% of the total body of the TD literature. Further insights on the current research gaps are documented in the following section.

1.3.1 Technical Debt Research Gaps

Regarding the current research gaps of TD, a first consideration has to be made on understudied or inexplored TD types. Fortunately, from the secondary studies considered, we can evince that the results regarding the less explored areas of TD are consistently reported in the literature [10, 11, 9, 4, 5]. From such results, we can evince that the following areas of research result to date to be only mentioned or marginally explored: *People TD*, *Requirements TD*, *Documentation TD*, *Service TD*, and *Versioning TD*.

A refined level of granularity can be considered by examining the decomposition of TD types into subdomains (as described by Li et al. [9]). By following this additional decomposition it is possible to assess which subdomains of the most studied research categories result to be understudied or exhibit research gaps. Regarding *Code TD*, such TD type results to date to be the most explored, and all of its subdomains, e.g. code duplication and coding violations, result to be considered in a relative high number of papers (between 7 and 20 researches per subcategory [9, 11]). By considering instead *Design TD*, several subcategories result to be only marginally explored. In particular, dealing with incomplete design specification and its relative impact on TD results to be a topic that, while mentioned in the literature, remains to date unexplored. Similarly, architectural antipatterns are known to have a negative impact on the TD of a software-intensive system, but their precise identification, measurement and management remains to date unknown. Finally, in the context of testing, deferring testing[2] results to be cited in the literature as a source of TD, but no studies focusing on such aspect can (to the best of our knowledge) be found.

By considering the most active TD research topics, we can identify also some common subcategories that result to be explored the least for each type of TD. If we consider the top three researched TD types, i.e. *Code TD*, *Design TD* and *Architecture TD*, we can notice that all of these categories lack prioritization methods aimed to rank TDI items according to their relevance. In addition, monitoring of TD items throughout the evolution of software-intensive systems results to date to be a vastly understudied topic at both code, design and architectural level. Finally, as pointed out by Alves et al. [4], the TD research filed severely lacks studies focusing on representation and documentation of TDIs. Such lack of research can be found at each of the previously introduced levels of abstraction, and results to have a deep impact on researchers and practitioners. In fact, without means to comprehensively report and present TDI analyses, it is hard to convey TD related information to people that are unfamiliar

[2] i.e. a bug fixing process that results to be associated to a negative return of investment (ROI)

with the topic. This results in a reduced awareness of researchers and practitioners alike, leading to the unconscious introduction of TDIs throughout the life cycle of a software product.

1.3.2 Further Technical Debt Research Opportunities

From a more careful inspection of the literature several other shortcomings of the TD research can be consistently spotted in the literature. A common trend that emerges is the lack of efficient tools with which TDIs can be identified, measured, monitored and eventually refactored. This demonstration of lack of tools can be found in a number of heterogeneous TD related researches, spanning over different TD activities and TD types. The problem emerging from the lack of TD related tools is particularity rooted in industry, where a steep-learning curve and upfront investment often discourages practitioners into investigating TD related issues.

In addition to the lack of tools, another recurrent topic throughout the literature is the scarcity of industrial case studies. In fact, most of the empirical researches focus on synthetic artifacts measured in controlled environments. This could potentially lead to a major threat to external validity, which could cause a divergence of the gathered results from the actual ones that could be collected by considering software-systems "in the wild". In addition, most of such "toy" artifacts are code based, as other artifacts (such as documentation) is harder to create artificially. To date only a few studies were base on actual documentation [40, 20]. This is also due to the fact that, as hinted to in Section 1.3.1, most of the studies building up the TD body of literature consider source code as measurand for their experiments.

Another aspect of TD research that requires to be further investigated is the accurate estimation of TD *principal* and *interest* values. In fact, while these concepts are widely accepted in the research community, to date no accurate method has been conceived in order to precisely calculate these two values. Hence it is difficult to estimate the concrete impact of a TDI and the ROI associated to refactoring a TDI in the present or future time.

1.4 State of the Art Overview

From the inspection of the TD body of literature the following take away messages can be evinced:

- TD results to be an active research field, characterized by a growing interest of researches and practitioners;
- The researchers focus on different types of TD; 12 such types were identified in the literature, as reported in Table 1.1;
- The vast majority of TD studies focuses on source code related TD;
- *Architectural TD* and *Design TD* result to be the second and third most studied levels of abstraction;
- Several types of TD, e.g. *Requirements* and *Documentation TD*, result to be only marginally explored;
- Many sub-fields of the most studied TD types result to be only mentioned in the literature;
- Prioritization, monitoring, visualization and documentation of TDIs result to be understudied TD activities;
- TD related research seem to severely lack tools for automated TDI identification, measurement and monitoring;
- Only very seldom researches are supported by industrial case studies: The vast majority considers synthetic artifacts measured in a controlled environment;
- An accurate model to calculate *principal* and *interest* values is currently not available.

1.5 Architectural Technical Debt

As described in the previous section, several types of TD and potential research opportunities can be identified in the literature. A particularly interesting and promising research area result to be architectural technical debt, i.e., debt incurred at the architectural level. As described by Van Vliet [41], between 50% and 70% of resources of a software project are allocated to maintenance processes. If such processes are neglected or not correctly carried out, software-intensive systems tend to slowly deteriorate through time, potentially leading to an obsolete or even failing system. During the development phases, software architecture plays a crucial role in the implementation of software systems [8]. Hence, if debt at the architectural level is neglected, it can introduce TDIs that have a tremendous impact on the overall TD

of software-centric systems. Hence, guided by industrial relevance and research potential, we opt to focus the studies reported in this thesis on investigating ATD phenomena. In general terms, ATD is referred to sub-optimal decision taken at the architectural level, which usually result in the conceivement of immature software architectural artifacts [11]. Such ill-suited architectural decisions can be of different types, e.g. they can be implicit or explicit [42], and can be made consciously or unconsciously [43]. In the literature, ATDIs are classified in four main categories [11], namely:

- *Architectural dependencies:* Ill-suited dependencies between software components of a system, including module dependencies, external dependencies, and external team dependencies [44];
- *Non-uniformity of patterns and policies:* ATDIs related to violations of naming conventions, non-uniform design, and violation of architectural patterns [45];
- *Code related:* Some authors associate various code related issues to ATDIs [44, 46, 47, 48], e.g., lack of code documentation, high code complexity and code duplication.
- *Quality Attribute (QA):* ATDIs which are strongly related to the implementation or test of non-functional properties, especially maintainability and evolvability [45, 49].

To date, the question regarding how to accurately identify and measure the magnitude and impact of ATDIs is still open. In particular, many ATD related studies result to be characterized by threats of external validity. This leads in many cases in the output of context specific results, which are difficult to apply to other case studies. While it might be difficult to totally abstract from some context specific aspects, e.g., the implementation language adopted, documentation format utilized etc., some level of abstraction is required in order to identify commonalities between different ATD instances. In such a way, it is possible to detect common trends that might be at the root of the identification of ATDIs.

Due to its abstract nature, and scarcity of tools available, ATD is regarded by Kruchten et al. as one of the most challenging TD types to be undercovered [8]. Most of the current methods aiming to identify ATDIs rely on interviews with software architects, or summary code dependency analyses (e.g., violation of modularity). In order to further advance the identification of ATDIs it is crucial to carefully evaluate what

information of software systems is already available, and how this can be used to detect ATDIs.

1.6 Research Goal and Research Questions

In this thesis, we aim at progressing the knowledge on Architectural Technical Debt, with the precise intent of understanding if, and in affirmative case how, we can conceive accurate approaches to identify and measure architectural technical debt items present in software-intensive systems.

Specifically, we formulate the "master" research question (RQ) of this thesis as follows:

RQ: What strategies can be used to identify and manage architectural technical debt?

To answer comprehensively this research question, we divide it into a set of sub-research questions (referred from now on as "research questions" for the sake of conciseness, also depicted in Figure 1.3).

Specifically, in order to devise strategies aimed at identifying and managing ATD, we require a sound knowledge of the approaches conceived to identify ATD which have been designed so far. Hence, we formulate our first research question as follows:

RQ1: What is the state of the art of architectural technical debt identification?

The answer to RQ1 enables us to gain knowledge on ATD in order to devise techniques to identify ATD. Specifically, we can leverage and combine existing methodologies, tools and findings, in order to obtain a high-level overview of the ATD present in software-intensive systems. Among the different dimensions of ATD, we opt to focus this work on one of the most explored dimensions, namely source code related ATD, in order to leverage already established findings in the field. Our second research question is hence formulated as follows:

RQ2: How can design issues detectable by tools be used to gain an overview of the architectural technical debt residing in a software system?

Answering RQ1 allows us to identify which ATD-related research topics with industrial relevance are only marginally investigated. We hence involve in our investigation

industrial practitioners, bringing closer the often mentioned gap between academic research and industrial practices. Ultimately, we aim at building a theory, grounded on the knowledge of experienced practitioners, gravitating around the concept of ATD. The research questions which guides this research topic states:

RQ3: What is architectural technical debt according to software practitioners?

The research outlined in research questions RQ1-RQ3 results to be (to a vast extent) technology agnostic, and adopts a “top-down", observatory, research strategy. Nevertheless, it is also possible to steer our endeavors to investigate ATD of specific development contexts, by utilizing a “bottom-up", data-driven, research approach.

From RQ1, we evince that, despite its current relevance, to date no study has been conducted on ATD present in the Android ecosystem, the most popular mobile ecosystem to date[3]. In addition, from RQ2 and RQ3 we also note that ATD can vary greatly according to the specific technological context considered. In order to gain further insights into the ATD phenomenon, we hence decide to focus in the second part of this thesis on the Android ecosystem. Specifically, we open this research topic by assessing the evolution of one of the most prominent quality attributes impacted by TD, namely maintainability. This provided us a better understanding of the context at hand, on which Android-specific ATD studies could be based. The RQ which marks the beginning of this research topic states:

RQ4: How are maintainability issues of Android applications characterized?

The next step following RQ5 is to conceptualize a methodology targeting the detection of ATD in Android applications. This research consists in the formulation of a high-level approach conceived for the automatic or semi-automatic detection of ATD of Android applications. The research question leading such project is formulated as follows:

RQ5: How can architectural technical debt items be identified in Android Applications?

In the next chapter, we briefly present the research methodologies utilized to answer our research questions. Such methodologies, and their specifics, are further documented in the section regarding the study design of the subsequent chapters.

[3]https://gs.statcounter.com/os-market-share/mobile/worldwide

1.7 Research Methodology

Given the multifaceted nature of the research questions documented in the previous section, we adopt for this thesis distinct research techniques. Specifically, we use as foundation for our investigations empirical methodologies applied to the field of software engineering, which are presented in Table 1.2 and further detailed in the remainder of this section.

Table 1.2 – Overview of research method per study

Study	Chapter	Method	Quantitative Analysis	Qualitative Analysis	Practitioners Involvement
ATD state of the art	2	SLR	✔	✔	
ATD index	3	EE, S	✔		✔
ATD theory	4	GT, FG		✔	✔
Android issues	5	CE	✔		
Android guidelines	6	MMES	✔	✔	✔

Methods. SLR: Systematic Literature Review; CE: Controlled Experiment; EE: Empirical Evaluation; S: Survey; GT: Grounded Theory; FG: Focus Group; MMES: Mixed-Method Empirical Study.

- *Systematic Literature Review (SLR)*: This research method is evidence-based technique which *de-facto* constitutes one of the most adopted techniques to systematically inspect the literature on an already established research topic. This approach enables us to gain an objective stance in order to systematically filter the existing academic literature, conduct data extraction and analysis, and draw conclusion on the obtained data. The systematic literature review is adopted in order to answer RQ1.
- *Controlled Experiment (CE):* This method entails the investigation of a particular phenomenon observed in a controlled environment. This technique, utilizing both qualitative and quantitative analyses, is best fitted to carry out studies with a narrow focus, and provides detailed results which are often used to validate previous findings or lay the groundwork for future research endeavors. We take advantage of case studies in order to answer RQ4.
- *Grounded Theory (GT)*: This technique, borrowed from the sociology research field, constitutes an established process to build a theory revolving around a specific topic. The resulting theory is established by grounding its concepts in evidence-based data gathered from sources of various nature. Specifically, we choose to ground our theory on the personal knowledge of experienced software engineering practitioners. We adopt this approach in order to answer RQ3.

- *Focus Group (FG)*: This method concisely entails conducting interviews with groups of individuals sharing a common background knowledge. In contrast to individual interviews, FGs allow participants to compare their experiences, jointly discuss opinions on it, and release potential inhibitions with respect to a discussed phenomenon. In this thesis, we used FG to evaluate our grounded theory by leveraging the criteria characteristic of the Glaserian GT method, *i.e.*, the method that was used to construct the theory.
- *Mixed-method empirical study (MMES)*: In certain instances, it is possible to leverage and combine different empirical methodologies in order to achieve a comprehensive answer for a research question. In our case, we use a blend of research methods, namely a multivocal literature review, semi-structured interviews with practitioners, and grounded theory techniques, in order to answer our last research question, RQ5.

1.8 Outline and Contribution

1.8.1 Thesis at a Glance

An overview of the research reported in this thesis is documented in Figure 1.3.

The first step consists of a systematic literature review on ATD, which was presented at the first edition of the ICM/IEEE International Conference on Technical Debt (TechDebt) in 2018 [50]. Among other results, from the literature study (RQ1) we are able to collect data in order to (i) gather intelligence on which software artifacts and methodologies can be leveraged to identify and manage ATD, and (ii) compare the state of the art of academic research with industrial practices.

After a comprehensive review of the state of the art of academic research, we progress with the formulation of an index, ATDx, which leverages statically detected SonarQube[4] issues in order to gain a high-level overview of the health of software systems w.r.t. ATD (RQ2), and the establishment of a grounded theory on ATD (RQ3).

Subsequently, we conduct a deep dive into Android-related ATD. This topic opens with a “bootstrap" study investigating maintainability issues of Android applications [51]. This study, blossoming from the industrial partnership with the Software Improvement Group (SIG), characterizes the maintainability issues in which Android developer incur, and is carried out by conducting a large-scale analysis of real-world Android applications (RQ4). The results of such study lay the groundwork for our future endeavors. Subsequently, we present an approach, based on guideline extraction, architecture reverse-engineering, and compliance checking, targeting the automatic identification of Android specific ATD hotspots (RQ5) [52, 53], which

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

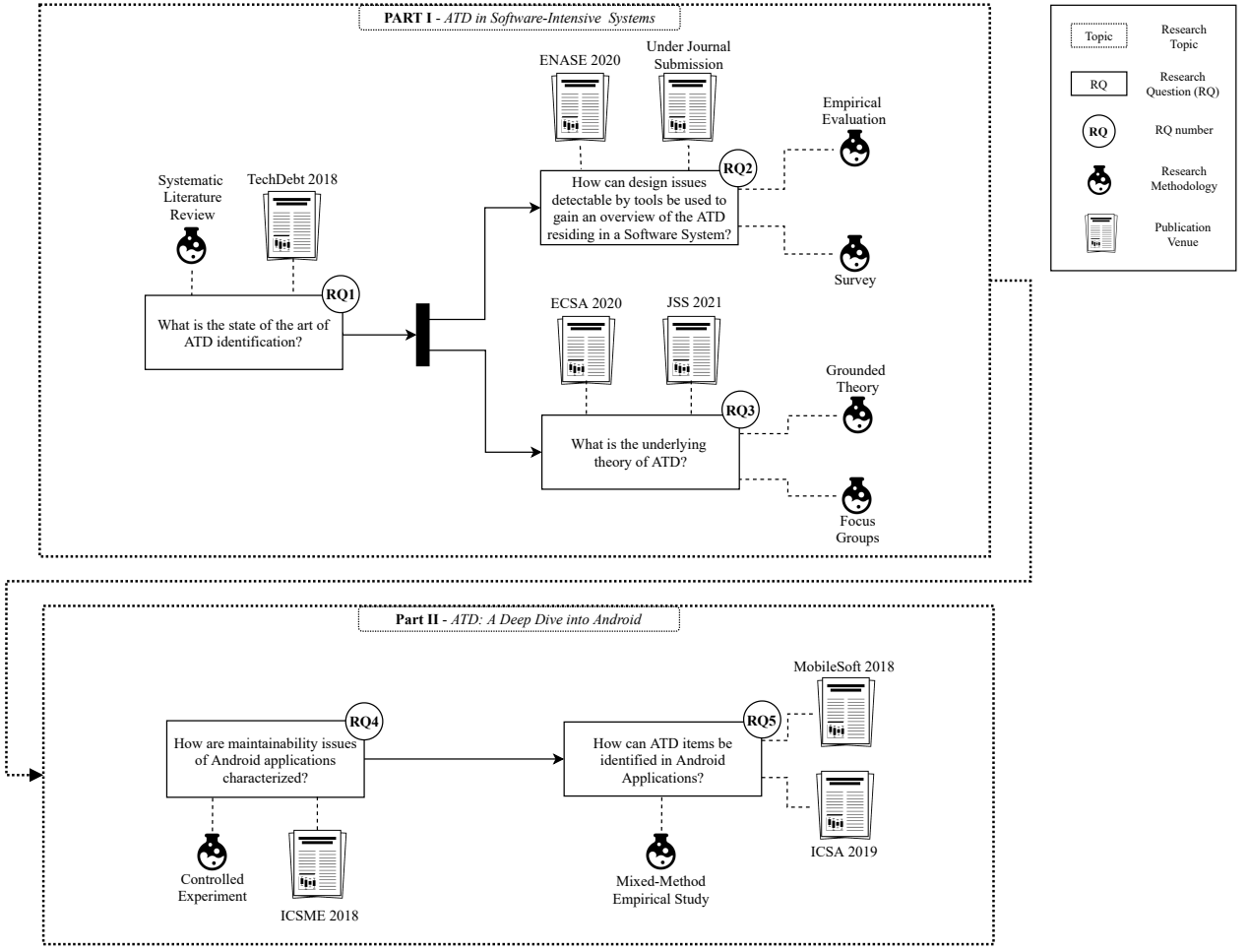


Figure 1.3 – Thesis at a glance: Research questions, methodologies and publication venues

constitutes the closing chapter of the the ongoing research presented in this thesis.

1.8.2 Authorship overview

In this section, we provide an overview of the contributions presented in each chapter of the thesis.

- **Chapter 1:**
  - **R.Verdecchia** *"Identifying Architectural Technical Debt: Moving Forward*, International Conference On Software Architecture (ICSA), 2018 [1].
    The candidate was the sole contributor of the work and paper writing.
- **Chapter 2:**
  - **R.Verdecchia**, I.Malavolta, and P.Lago "*Architectural Technical Debt Identification: The Research Landscape*", International Conference on Technical Debt (Techdebt), 2018 [50].
    The candidate was the main contributor of the work and paper writing. The remaining authors edited and supervised the written work.
- **Chapter 3:**
  - **R.Verdecchia**, P.Lago, I.Malavolta, and I.Ozkaya "*ATDx: Building an Architectural Technical Debt Index*", International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE), 2020. [54].
    The candidate was the main contributor of the work and paper writing. The remaining authors edited and supervised the written work.
  - **R.Verdecchia**, I.Malavolta, P.Lago, and I.Ozkaya "*Empirical Evaluation of an Architectural Technical Debt Index in the Context of the Apache and ONAP Ecosystems*", Journal article under submission [55].
    The candidate was the main contributor of the work and paper writing. The remaining authors edited and supervised the written work.
- **Chapter 4:**
  - **R.Verdecchia**, P.B.Kruchten, and P.Lago "*Architectural Technical Debt: A grounded Theory*", European Conference on Software Architecture (ECSA), 2020 [56].
    The candidate was the main contributor of the work and paper writing. The remaining authors edited and supervised the written work.
  - **R.Verdecchia**, P.B.Kruchten, P.Lago, and I.Malavolta "*Building and Evaluating a Theory on Architectural Technical Debt in Software Intensive Systems*", Journal of Systems and Software (JSS), 2021 [57].
    The candidate was the main contributor of the work and paper writing. The remaining authors edited and supervised the written work.

- **Chapter 5**

  I.Malavolta, **R.Verdecchia**, M.Bruntink , B.Filipovic and P.Lago"*How Maintainability Issues of Android Apps Evolve*", International Conference on Software Maintenance and Evolution (ICSME), 2018 [51].
  The candidate worked on the design of the study, data extraction, data analysis, and paper writing. I.M. participated across all phases of the study. M.B. provided the tool and the infrastructure for the execution of the study. B.F. is the Master student who built the dataset and piloted the first version of the study under the supervision of I.M. P.L. edited and supervised the written work.

- **Chapter 6**

  R.Verdecchia "*Identifying Architectural Technical Debt in Android Applications through Compliance Checking*", International Conference on Mobile Software Engineering and Systems (MobileSoft), 2018 [52].
  The candidate was the sole contributor of the work and paper writing.

  R.Verdecchia, I.Malavolta, and P.Lago "*Guidelines for Architecting Android Apps: A mixed-method Empirical Study*", International Conference on Software Architecture (ICSA), 2019. [53].
  R.V. was the main contributor of the work and paper writing. The remaining authors edited and supervised the written work.

1.8.3 Other contributions

In this section we provide a brief overview of other studies, not part of this thesis, which were carried out during the years of the Ph.D. devoted to research activities, and the relative publications such studies let to.

Technical Debt
Researches investigating topics related to technical debt not included in the thesis.

- S.Ospina, **R. Verdecchia**, I. Malavolta, P. Lago, *ATDx: A tool for Providing a Data-driven Overview of Architectural Technical Debt in Software-intensive Systems*, European Conference on Software Architecture (ECSA), 2021 [58]
- J. Bogner, **R. Verdecchia**, I. Gerostathopoulos, *Characterizing Technical Debt and Antipatterns in AI-Based Systems: A Systematic Mapping Study*, International Conference on Technical Debt (TechDebt), 2021 [59]
  Best Presentation Award
- **R. Verdecchia**, P. Lago, I. Malavolta, I. Ozkaya, *ATDx: Prototype Implementation Technical Report*, VU Technical Reports, 2020 [60]

Software Energy Efficiency
Researches studying the identification and alleviation of energy-greedy portions of source code present in software-intensive systems.

- **R. Verdecchia**, A. Guldner, Y. Becker, E. Kern, *Code-Level Energy Hotspot Localization via Naive Spectrum Based Testing*, Advances and New Trends in Environmental Informatics - Managing Disruption, Big Data and Open Science, 2018 [61]
- **R. Verdecchia**, R. A. Saez, Giuseppe Procaccianti, and P. Lago, *Empirical Evaluation of the Energy Impact of Refactoring Code Smells*, International Conference on Information and Communication Technology for Sustainability, (ICT4S), 2018 [62]
  🏆 *Runner-Up Best Paper Award*
- **R. Verdecchia**, Giuseppe Procaccianti, I. Malavolta, P. Lago, and J. Koedijk, *Estimating Energy Impact of Software Releases and Deployment Strategies: The KPMG Case Study*, ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 2017 [63];
- P. Lago, **R. Verdecchia**, N. Condori-Fernandez, E. Rahmadian, J. Sturm, T. van Nijnanten, R. Bosma, C. Debuysscher, and P. Ricardo, *Designing for Sustainability: Lessons Learned from Four Industrial Projects*, Advances and New Trends in Environmental Informatics, 2020 [64]
- **R. Verdecchia**, P. Lago, C. de Vries, *The LEAP Technology Roadmap: Lower Energy Acceleration Program (LEAP) Solutions, Adoption Factors, Impediments, Open Problems, and Scenarios*, VU Technical Reports, 2021 [65]

Software Testing
Researches proposing scalable approaches based on clustering and ranking techniques to solve prominent regression testing problems.

- **R. Verdecchia**, E. Cruciani, B. Miranda, A. Bertolino, *Know Your Neighbor: Fast Static Prediction of Test Flakiness*, IEEE Access, 2021 [66]
- F. Coró, **R. Verdecchia**, E. Cruciani, B. Miranda, A. Bertolino, *JTeC: A Large Collection of Java Test Classes for Test Code Analysis and Processing*, International Conference on Mining Software Repositories (MSR), 2020 [67]
- E. Cruciani, B. Miranda, **R. Verdecchia**, A. Bertolino, *Scalable Approaches for Test Suite Reduction*, ACM/IEEE International Conference on Software Engineering (ICSE), 2019 [68]
  🏆 *ACM SIGSOFT Distinguished Paper Award.*
- B. Miranda, E. Cruciani, **R. Verdecchia**, A. Bertolino, *FAST Approaches to Scalable Similarity-based Test Case Prioritization*, ACM/IEEE International Conference on Software Engineering (ICSE), 2018 [69]

Software Architecture Education and Training
Studies proposing serious-games to educate and train in software architecture concepts and principles.

- P. Lago, J. F. Cai, R. C. de Boer, P. Kruchten, and **R. Verdecchia**, *DecidArch: Playing Cards as Software Architects*, Hawaii International Conference on System Sciences (HICSS), 2019 [70]
  🏆 *Best Paper Award.* 🏆 *ISSIP-IBM-CBA Student Paper Award for Best Industry Studies Paper.*
- P. Lago, Jia F. Cai, Remco C. de Boer, P. Kruchten, and **R. Verdecchia**, *DecidArch v2: An improved Game to teach Architecture Design Decision Making*, International Workshop on decision Making in Software ARCHitecture (MARCH), 2019 [71]

Part I

Architectural Technical Debt in Software-Intensive Systems

2 Architectural Technical Debt Identification: The Research Landscape

The good news about computers is that they do what you tell them to do. The bad is that they do what you tell them to do.

Ted Nelson

This chapter is based on R. Verdecchia, I. Malavolta, and P. Lago, *Architectural Technical Debt: The Research Landscape*, International Conference on Technical Debt (TechDebt), 2018 [50].

Chapter 2. Architectural Technical Debt Identification: The Research Landscape

This chapter targets the first research question of this thesis (RQ1). Specifically, in this chapter we document a Systematic Literature Study (SLR) on architectural technical debt identification. The SLR is specifically conceived to lay the sound basis necessary in order to answer the subsequent research questions of this thesis. The goal of the investigation presented in this chapter is to identify, classify, and evaluate the state of the art on ATD identification from the following three perspectives: publication trends, characteristics, and potential for industrial adoption. Specifically, starting from a set of 509 potentially relevant studies, we systematically selected 47 primary studies and analyzed them according to a rigorously predefined classification framework. The analysis of the obtained results will support both researchers and practitioners by providing (i) an assessment of current research trends and gaps in ATD identification, (ii) a solid foundation for understanding existing (and future) research, and (iii) a rigorous evaluation of its potential for industrial adoption.

Contents

2.1 Introduction

In order to bootstrap our investigation on architectural technical debt identification, a required first step is to gain a rigorous overview of the state of the art. Hence, in this opening chapter, we present a systematic literature review specifically studying the existing body of knowledge on architectural technical debt identification. While some studies have been conducted to provide an overview of the state of the art on ATD in general [11, 72], none of these focused specifically on ATD identification. The **goal** of this study is to fill this gap by providing an evidence-based overview of the existing ATD identification research landscape. This study has been carried out by adopting a well-established **methodology** called systematic mapping [73, 74], and we applied it on peer reviewed papers focusing on ATD identification in software-intensive systems. Through our systematic mapping process, we selected 47 primary studies among 509 potentially relevant studies, fitting at best a set of rigorously-defined inclusion and exclusion criteria. Then, we created a dedicated classification framework composed of 13 different parameters for comparing techniques for ATD identification, and we applied it to all primary studies. We analyzed and discussed the obtained data under three complementary perspectives: publication trends, characteristics, and potential for industrial adoption.

The main **contributions** of this study are the following:

- an objective *map* of the state of the art in ATD identification;
- a rigorously-built *classification framework* for past, present, and future techniques for ATD identification;
- an *evaluation* of publication trends, characteristics, and potential for industrial adoption of existing research on ATD identification;
- a *discussion* of the emerging research trends, patterns, and gaps, and their implications for future research on ATD identification.

The **audience** of this study is composed of both (i) *researchers* willing to contribute to the area of ATD identification (ii) *practitioners* willing to understand existing research on ATD identification and adopt the most appropriate solutions for their technical, business, and organizational needs.

The remainder of this chapter is organized as follows. Section 2.2 discusses related work and compares it to this study. Section 2.3 presents the design of this study from a methodological perspective, whereas Sections 2.4, 2.5, and 2.6 discuss obtained results. Finally, Section 2.7 describes threats to validity and Section 2.8 closes the chapter.

2.2 Related work

A number of secondary studies focusing on TD exist to date. In Table 2.1 we give a schematic overview of those studies and in the following we discuss each of them.

Table 2.1 – Secondary studies on TD

Secondary study title	Year	Focus	#Studies	Time frame
Tom et al. [10]	2013	TD	35	N/A
Li et al. [9]	2015	TD	94	1992-2013
Alves et al. [4]	2015	TD	100	2006-2014
Ampatzoglou et al. [5]	2015	TD (financial)	69	2009-2013
Besker et al. [11]	2016	ATD	26	2012-2015
Besker et al. [72]	2017	ATD	42	2011-2016
This study	**2018**	**ATD identification**	**47**	**2009-2017**

Most of the secondary studies on TD consider it only from a general perspective (i.e., they are not specific to architectural TD). For example, Li et al. [9] consider a set of 94 primary studies, with the goal to provide a comprehensive understanding of the notion of TD and the related research activities. This was achieved by thematically classifying the primary studies into 10 coarse-grained TD types, among which ATD results to be the second most studied subject (together with test TD and design TD), appearing in 25 different researches.

In a similar research carried out by Alves et al. [4], the TD literature was inspected to characterize the types of TD, their indicators, management strategies, maturity level, and possible visualization techniques. The authors analyzed 100 studies published between 2010 to 2014, which resulted in the conception of a preliminary taxonomy of TD types and an overview of the current research trends of TD. While, as reported also in the research by Li et al. [9], ATD resulted to be the second most frequent TD type, no in-depth analysis was reported for such topic.

In an earlier research, Tom et al. [10] present a literature review considering 35 primary studies. The focus in this case is to identify the nature of TD and its impact on software development activities. From the results, the authors derived a theoretical framework illustrating the dimensions of TD, attributes, precedents and outcomes. As compared to the previously-mentioned literature studies, the focus of this research is to gain understanding of the current research activities related to TD; accordingly, it does not concentrate on any specific aspect of TD.

In their systematic literature review, Ampatzoglou et al. [5] considered a more detailed overview of a specific TD topic, namely the economic implications of TD. In their work, the authors

analyzed 69 primary studies to understand how financial aspects are defined in the TD context and how these are related to various aspects of software engineering. Their results show that financial approaches for TD management lack consistency in their applications, as the same approach is utilized differently in different studies. In [5], ATD is considered exclusively from a financial point of view.

The secondary study of Besker et al. [11] is the closest to ours by focusing exclusively on ATD-related literature. The authors inspect 26 primary studies to conceive a descriptive model aimed to provide a comprehensive interpretation of the ATD phenomenon. Their model identifies the main characteristics of ATD in four clusters: *ATD Identification*, ATD Checklist, ATD Impediments, and ATD Management. Each cluster is further decomposed into Focus areas (e.g. Relevance, Challenges, Methods/Tools), which are characterized by different aspects (e.g. Methods/tools is composed of Measuring, Tracking and Evaluating). This study was extended in a later publication [72] presenting a more comprehensive investigation of the literature and an in-depth analysis of the results. Our study differs from theirs by zooming into a specific challenge they identify, namely *ATD identification*.

In conclusion, by inspecting the TD secondary studies, we can observe that none of the studies aims directly at the characterization of existing approaches for ATD identification. We therefore conducted this research, focusing exclusively on the research landscape of ATD identification, in order to fill this gap and complement the existing literature.

2.3 Study Design

In this section we report the study design that was strictly followed while planning and conducting research reported in this chapter. The study design was concieved by following a set of well-established guidelines for software engineering literature studies [73].

2.3.1 Research Goal

The research was designed with the goal of characterizing comprehensively the current state of the art of ATD identification research. More specifically, by following the Goal-Question-Metric approach [75], our goal can be formalized as follows:

Purpose	Identify, classify, and evaluate
Issue	publication trends, characteristics, and potential for industrial adoption
Object	of existing techniques for ATD identification
Viewpoint	from the researcher's and practitioner's point of view.

2.3.2 Research Questions

From our research goal, we can derive the following three research questions underlying our study:

RQ1.1: *What are the **publication trends** about techniques for ATD identification?* By answering this research question we aim to assess the ongoing trends of scientific interest on ATD identification techniques in terms of publication frequency, most prominent venues where academics are publishing their results on the topic and most recurrent venue types.

RQ1.2: *What are the **characteristics** of existing techniques for ATD identification?* By answering this research question we aim at providing (i) a solid framework for examining and classifying existing (and future) research on ADT identification techniques, and (ii) an understanding of current research trends and gaps in the state of the art of such techniques.

RQ1.3: *What is the **potential for industrial adoption** of existing techniques for ATD identification?* By answering this research question we aim at assessing to what extent the current ATD identification research results are ready to be transferred and adopted in an industrial context.

2.3.3 Search and selection

The search and selection process was designed as a multi-stage process, as depicted in Figure 2.1. This enabled us to rigorously control on the number and characteristics of the studies considered during the various stages. A description of each process followed is provided in the remainder of this section.

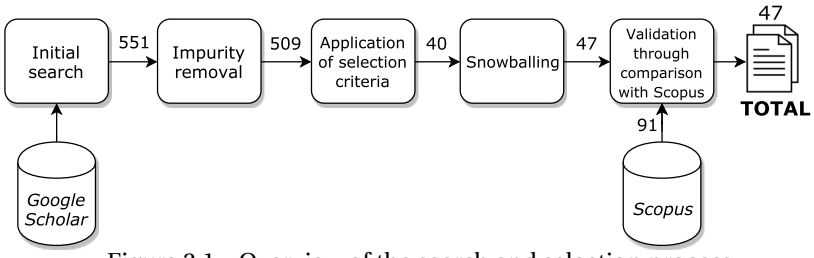


Figure 2.1 – Overview of the search and selection process

2.3.3.1 Initial search

To identify the initial set of studies we performed an automatic search query on one of the largest and most complete scientific database and indexing system, namely *Google Scholar*.

We selected such digital library for the following reasons: (i) it resulted to provide the highest number of potentially relevant studies compared to other four relevant libraries (Scopus, ACM Digital Library, IEEE Explore, and Web of Science), (ii) as reported in the set of guidelines by Wholin et al. [76], the adoption of such indexer results to constitute a sound choice to identify the initial set of literature for snowballing processes, (iii) the query results could be automatically extracted from the indexer. The research query utilized was conceived to encompass as much relevant studies as possible and is as follows:

Listing 2.1 – Search Query

```
1 TITLE:(architecture OR architectural OR architect OR
2 architecting OR TD OR "technical debt" OR ATD)
3 AND (architecture OR architectural OR architect
4 OR architecting) AND ("technical debt")
```

The query selects studies containing either a keyword referring to “architecture” or “technical debt” and related acronyms in their title (Listing 6.1, lines 1-2). Additionally, the full-text of the studies must contain both on of the keywords referring to “architecture” and the phrase “technical debt” (Listing 6.1, lines 3-4). The exclusive presence of related acronyms, i.e. “TD” and “ATD” in the full-text is not considered as a valid hit. The considered timeframe end was delimited by when the query was first executed (November 2017), in order to avoid potential discrepancies of results due to different query execution times. The start date was not set, in order to a void the introduction of a potential bias, even if it could have been set to when the term “technical debt” was first referenced [3].

2.3.3.2 Impurity removal

From the initial execution of the search query, a number of elements resulted not to be research papers (e.g. patents, standards etc.). Such occurrences were manually removed from the initial set of potentially relevant studies.

2.3.3.3 Application of selection criteria

Subsequent to the impurity removal process, we filtered all the remaining research papers according to a set of rigorously defined selection criteria. A research paper was included in the set of primary studies exclusively if it satisfied *all* of our inclusion criteria and none of the exclusion ones. Several exclusion rounds were adoptive by utilizing an adaptive reading depth [77], in order to thoroughly examining the literature in a time-efficient and objective manner. The inclusion and exclusion criteria utilized were:

I1- Studies focusing on TD identification in software-intensive systems. This inclusion

criterion is utilized to select exclusively studies considering TD.

I2- Studies focusing on the architecture of software-intensive systems. This inclusion criterion is utilized to filter out studies considering other levels of abstraction, such as specific code implementation details.

I3- Studies presenting or using a technique aimed to the identification of ATD in software-intensive systems. With this inclusion criteria, we ensure that only papers discussing the identification of ATD are included.

E1- Secondary or tertiary studies (e.g., systematic literature reviews, surveys, etc.). This exclusion criterion is adopted in order to exclude studies which do not report the desired level of detail of ATD identification techniques.

E2- Studies in the form of editorials and tutorial, short papers, and poster, as they are deemed to not provide the required level of detail and information.

E3- Studies that have not been published in English language, as their analysis would result to be too time consuming.

E4- Studies that have not been peer reviewed, in order to ensure the high quality of the studies considered.

E5- Duplicate papers or extensions of already included papers, in order to avoid possible threats to conclusion validity.

E6- Papers that are not available, as we cannot inspect them.

2.3.3.4 Snowballing

In order to mitigate potential biases w.r.t. the construct validity of this study, the automatic search was complemented by a snowballing process [78]. Specifically, a closed recursive backward and forward snowballing activity was conducted [76]. During this process, researches either citing and cited by the ones selected in the previous stages were examined, in order to enlarge the set of potentially relevant studies.

2.3.3.5 Validation through Scoups

In order to further mitigate the potential threats to validity resulting from the selection of a specific digital library, we conducted an supplementary search query execution on Scopus. We chose this additional database as it is defined as the largest abstract and citation database of peer-reviewed literature [79]. From the query execution, most of the paper indexed resulted to be either primary studies or papers excluded in the previous selection stages. Only 3 new papers were identified. After the application of the selection criteria, all were discarded.

2.3.4 Data Extraction

The purpose of this process was to (i) create a classification framework for the primary studies and (ii) extract the data from each primary study. The classification framework consists of three parts, each addressing one of the research questions (see Section 2.3.2).

2.3.4.1 Publication Trends (RQ1.1)

In order to assess the trends of ATD identification research, three attributes were considered, namely `publication year`, `publication type` and `publication venue`.

2.3.4.2 Research Focus (RQ1.2)

To characterize the focus of the primary studies a systematic keywording process [80] was adopted to define some of the parameters of our comparison framework. This process is constituted of two distinct steps: (i) *collection of keyword and concepts*, i.e. the identification of keywords and concepts by inspecting the full-text of all primary studies, and the subsequent combination of these to clearly identify the context, nature, and contribution of the research (ii) *keyword clustering*, i.e. the clustering the identified keywords and concepts into categories, in order to build up a classification framework. The output of this stage will be the classification framework containing all the identified parameters (each of them having a specific type and possible values), representing a specific aspect of ATD identification. The parameters considered in order to answer RQ2 were: `architectural level` (keyworded), `ATDI definition` (keyworded), `analysis type` (keyworded), `analysis input source` (keyworded), `temporal dimension`, `ATD resolution`, and `tool support`. For a definition of each attribute we refer to Section 6.3.4.2.

2.3.4.3 Potential for industrial adoption (RQ1.3)

To assess the potential for industrial adoption, four distinct facets were considered, namely: `tool availability`, `industry involvement`, `rigor` and `industrial relevance`. The data of the last two attributes were collected by adopting an *ad-hoc* data extraction process [81].

2.3.5 Data Synthesis

Through a data synthesis process we aggregated and summarized the data extracted from the primary studies [82, §6.5] in order to understand, analyze, and classify the landscape of

ATD research. In particular, we adopted content analysis (to categorize and code the primary studies in broad thematic categories) in combination with narrative synthesis (used to describe details and interpret the findings resulting from the content analysis).

2.3.6 Study Replicability

In order to provide the ability to fully replicate the research, a replication package of the study is publicly available[1]. The package includes (i) the protocol describing comprehensively the study design details, (ii) a detailed description of all the parameters composing the classification framework, (iii) the raw data extracted in each phase (iv) the scripts utilized for the data processing, and (v) the list of primary studies.

2.4 Results - Publication trends (RQ1.1)

In the remainder of this section we report the results of the analysis of publication year, type, and venue of each primary study. In the tables reported in this and the following sections (Section 2.4-2.6) the recurrence of concepts across primary studies is reported in both numbers and the proportional sizes of the pink horizontal bar charts (see Columns "#Studies" of Tables 2.2-2.10).

2.4.1 Publication year

Figure 2.2 shows the number of primary studies appearing each year, clustered by venue type. Primary studies span from 2009 to 2017, i.e., the year in which the search query was executed. While the term TD was first coined in 1992, we can observe that several years passed before the TD metaphor was explicitly considered in software architecture. Interestingly, the paper published in 2009 (P41) does not explicitly refer to "ATD" but considers "architectural bad smells" instead.
In general, a growing trend can be identified through the years, demonstrating the recent interest of researchers and practitioners in the subject. As an outlier, a drop of number of primary studies can be noticed for the year 2017. Such occurrence should be attributed to the fact that the search query used to select the primary studies was executed before the end of 2017, leading to partial results for this year.The reported publication trend is confirmed also in the recently published study by Besker et al. [72], focusing on the main ATD characteristics and relations among them.

[1] http://www.s2group.cs.vu.nl/techdebt-2018-replication-package

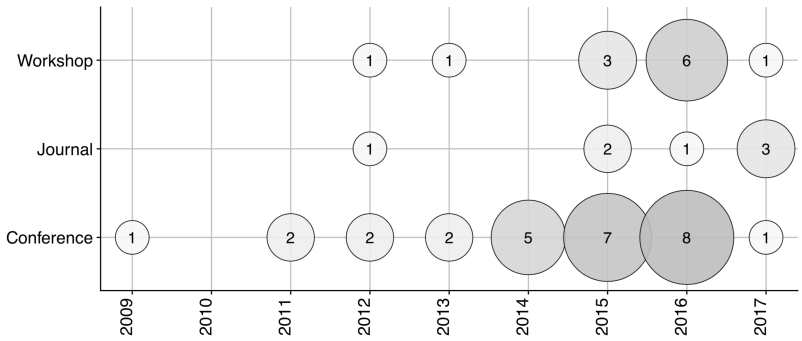


Figure 2.2 – Primary studies publication year and venue type

2.4.2 Publication types

From the distribution depicted in Figure 2.2 we observe that the majority of primary studies is published in conferences (28/47), followed by a non-negligible number of workshops (12/47). Only a modest number of studies (7/47) is published in journals. We can conjecture that such trend is due to the relatively recent interest in ATD identification by the software engineering research community. We can expect a growth in the number of more scientifically-rewarding publication types (like journals) in the future. This conjecture is partially confirmed by the 3 journal publications only in 2017.

2.4.3 Publication Venues

Table 2.2 reports on the number of studies appearing in the most recurrent venues. There we can observe that ICSE is the most frequent venue (9/47) followed by MTD (6/47) and ICSA/WICSA[2] (6/47). The presence of MTD as second most recurrent venue highlights the importance that such workshop has in the TD research area. In general, due to the nature of the most recurrent venues, we can conjecture that the primary studies are characterized by being of high quality and have potentially high interest and resonance in the scientific community.

Table 2.2 shows that a relatively high number of primary studies (18/47) have been published in different venues, spanning different research areas like software maintenance and evolu-

[2]From 2017, WICSA has been renamed as ICSA.

Table 2.2 – Publication venues

Venue acronym	#Studies	Studies
International Conference on Software Engineering (ICSE)	9	P1, P6, P10, P15, P20, P25, P32, P42, P45
Managing Technical Debt workshop (MTD)	6	P2, P21, P24, P30, P35, P40
International Conference Software Architecture (ICSA) / Working IEEE-IFIP Conference on Software Architecture (WICSA)	6	P7, P9, P11, P19, P22, P36
European Conference on Software Maintenance and Reengineering (CSMR)	3	P41, P46, P47
Information and Software Technology (IST)	2	P4, P23
International Conference on the Quality of Software Architectures (QoSA)	2	P5, P31
International Conference on Agile Software Development (XP)	2	P8, P28
Other	18	P43, P39, P27, P18, P38, P14, P17, P12, P29, P33, P34, P37, P3, P44, P16, P13, P26

tion (P14), software architecture erosion and architectural consistency (P3), and dependency and structure modeling (P27). This might indicate that the TD research community is still undergoing a consolidation phase.

Main findings (RQ1). ATD identification is attracting a growing scientific interest in the last years. The research landscape is quite fragmented, with ICSE, MTD, and ICSA as most targeted venues. So far, researchers mostly targeted conferences and workshops, but it is expected that journal publications will raise in the coming years.

2.5 Results - Research focus (RQ1.2)

Here we report the characteristics of ATD identification techniques as they emerged from our keywording process (see Section 2.3.4).

2.5.1 Level of abstraction

We uncovered four distinct and incremental levels of abstraction during our keywording process, namely: *Source Code Classes*, *Source Code Files*, *Source Code Packages*, and *Components and Connectors*.

As shown in Table 2.3, *source code packages*, defined as related files implementing the same functionality, are the most recurrent building blocks of the considered architectures (13/47). *Components and Connectors* are adopted by a similar number of studies (12/47), while *Source Code Classes* and *Source Code Files* are used in a lower number of studies.

Table 2.3 – Architecture level

Architecture keywording	#Studies	Studies
Source Code Packages	13	P3, P4, P5, P7, P9, P14, P27, P28, P30, P38, P39, P40, P47
Components and Connectors	12	P10, P11, P18, P21, P22, P23, P24, P34, P36, P41, P46, P47
Source Code Classes	9	P9, P24, P26, P28, P30, P31, P44, P45, P47
Source Code Files	8	P1, P15, P19, P20, P25, P32, P33, P42
Not specified	11	P2, P6, P8, P12, P13, P16, P17, P29, P35, P37, P43

A considerable number of primary studies (11/47) does not report explicitly the considered abstraction level. Those primary studies commonly rely on human knowledge (P6, P8, P12, P13, P29), self-admitted ATDI (P2, P16), and tools in which the architecture definition is implicit (P17, P35, P37, P43).

The obtained results suggest that as of today there is no common agreement in the literature about which level of abstraction should be considered when dealing with ATD identification. We can conjecture that this phenomenon may be a consequence of the fact that (i) a unique and well-accepted definition of software architecture is also missing in the state of the art and (ii) the level of abstraction in the proposed techniques is strongly influenced by the types of ATD sources, such as system source code, its architecture documentation, etc. (see Section 2.5.4 for the details on ATD sources).

2.5.2 ATDI Definition

Among the current definitions of technical debt, a widely adopted one was formulated by a group of experts during Dagstuhl seminar 16162 [83]. Simply referred to as the 16162 definition of technical debt, it specifies technical debt as *"design or implementation constructs that are expedient in the short term, but set up a technical context that can make a future change more costly or impossible"*. In this section we study ATD at a finer-grained level, by considering the definitions of ATD provided in the primary studies.

The keywording process resulted in four recurrent categories of ATDI (see Table 2.4): *Dependency Violations* among architectural components (27/47), *Non-modularity* (26/47), *Compliance Violations* (18/47) and *Change Proneness* (9/47). *Dependency Violations* describe architectural violations resulting from unfit dependencies among architectural components. Such type of ATDI is usually caused by unsound architectural design choices, incorrect implementation, or architectural deterioration. *Non-modularity* refers to the sub-optimal modularization of architectural components. Lack of modularity often causes small changes to propagate to other portions of a system, lowering the maintainability and evolvability of software systems. *Compliance violations* refer to the deviation w.r.t. a certain architectural pattern (e.g. model-view-controller) affecting the quality of the system. *Change Proneness*, instead, refers to architectural components that are modified with high frequency.

Table 2.4 – ATDI definition

ATDI keywording	#Studies	Studies
Dependency Violations	27	P1, P2, P7, P8, P9, P10, P11, P12, P14, P16, P17, P19, P20, P21, P22, P23, P27, P28, P30, P32, P34, P35, P38, P40, P41, P45, P46
Non-Modularity	26	P1, P2, P4, P5, P8, P9, P10, P11, P12, P14, P15, P16, P17, P18, P19, P20, P28, P29, P30, P32, P34, P35, P41, P42, P45, P46
Compliance Violations	18	P3, P4, P6, P7, P11, P12, P13, P21, P23, P24, P26, P31, P36, P37, P39, P43, P45, P47
Change Proneness	9	P1, P9, P14, P19, P20, P29, P32, P33, P45
Custom	18	P5, P8, P11, P16, P18, P19, P20, P22, P25, P28, P29, P30, P35, P37, P39, P41, P42, P44

In 25 primary studies more than one type of ATDI is used. In most of the cases this is due to the definition of more than one ATDI in the paper. For example, in P19 five different ATDIs related to *change proneness*, *dependency violations*, and *non-modularity* of source code files are defined. Additionally, a considerable number of papers (18/47) is based on a custom definition of ATDI that could not be mapped onto any specific category. Such occurrences focus on an ad-hoc definitions of ATD, e.g. lack of reusability (e.g., P9) or non-uniformity of package name patterns (e.g., P39).

From the high heterogeneity of the gathered data we can conclude that there is no common agreement on what defines an ATDI. We can hence conclude that, while different types of ATDI are required in order to comprehensively model ATD, the literature is still lacking in a comprehensive taxonomy for a sound classification of ATDIs.

Interestingly, while some papers mention the 16162 technical debt definition [83], we note that the definition of ATDI in the primary studies is never directly compared to such definition. In fact, the definition of ATD itself in primary studies is mostly implicitly defined *via* the description of the identification approaches. We conclude that more effort should be spent in clearly documenting the adopted definition of ATD and ATDI, and relate such definition with the other ones present in the literature for the sake of clarity and completeness.

2.5.3 Analysis Type

In Table 2.5 the most recurrent analysis types are reported. From the extracted data we can observe that a large number of analysis types are performed in the literature.

Table 2.5 – Analysis type

Analysis keywording	#Studies	Studies
Architectural Antipatterns and smells	25	P3, P4, P5, P7, P9, P10, P14, P15, P17, P19, P20, P24, P25, P26, P27, P28, P30, P32, P35, P37, P38, P41, P42, P46, P47
Modularity Analysis	19	P4, P5, P9, P10, P15, P17, P19, P20, P24, P28, P32, P33, P34, P35, P37, P41, P42, P45, P46
Evolution Analysis	16	P1, P4, P5, P6, P10, P15, P19, P20, P25, P26, P31, P32, P34, P38, P42, P45
Dependency Analysis	15	P14, P15, P22, P24, P27, P28, P30, P32, P33, P34, P35, P37, P40, P42, P43, P46
Cost Analysis	14	P4, P6, P8, P10, P17, P20, P21, P22, P27, P29, P30, P33, P34, P38
Human knowledge based	10	P6, P8, P11, P12, P13, P18, P21, P23, P29, P36
Compliance Checking	6	P3, P7, P31, P39, P43, P47
Change Impact Analysis	6	P9, P10, P11, P18, P21, P22
OO Relation Analysis	6	P1, P25, P26, P31, P44, P47
Visualization	5	P10, P21, P27, P40, P43
Manual Classification	4	P16, P39, P40, P41
Self Admitted	2	P2, P16

The majority of the ATD identification techniques are based on the identification of *architectural antipatterns and smells* among architectural components. Examples of identified architectural antipatterns include cyclic dependencies between architectural components and unstable Interfaces. In particular, such techniques usually entail (i) the identification of architectural components (usually achieved by clustering source code artifacts) and, (ii) the assessment of properties of/among the components, usually carried out through *modularity analyses* (19/47), *dependency analyses* (15/47), or a combination of the two[3].

Modularity analysis consists in assessing if system functionalities are separated into independent, self-contained modules [84]. Modularity resulted the second most recurrent type of analysis (19/47). Such approaches commonly entail the identification of architectural components and functional requirements of a software system, the evaluation of the modularization of components, and the eventual cost analysis of the rework cost required to carry out a refactoring process.

Dependency analysis is based on the evaluation of dependencies between architectural components in order to identify irregularities (e.g., circular dependencies). The use of Design

[3]We are aware that modularity and dependency analyses can be considered as particular types of architectural antipattern. Given their high frequency in the primary studies, we considered them as a separate categories.

Structure Matrices (DSM) for dependency analysis results to be quite widespread for ATDI identification (P1, P10, P15, P19, P20, P22, P27, P32, P42, P45).

In order to discover and understand ATD related phenomena, many studies rely on analyses of the *evolution* of software systems through time (16/47). Such approaches take as input historical data such as architectural documentation and version history. We observe that our primary studies adopt a heterogeneous set of approaches to identify ATD. For example, in P1 a technique based on locating co-changing files in order to identify "architectural roots" of systems is presented, while in P5 relates modularity metrics to the average number of modified components per commit. From the variety of approaches utilizing evolution analysis we can evince the high potential that historical data has for ATD identification.

A relatively high number of studies considers the financial aspect of managing ATD through *cost analyses* (14/47). Such studies range from risk analyses (e.g., P34) to methods aiding decisions on *if* and *when* ATD should be refactored (e.g., P6). The presence of numerous studies focusing on cost implications of ATD is a meaningful indicator. In fact this demonstrates how the ATD problem is rooted in practice and furthermore shows that the importance of effectively understanding ATD issues and planning future maintenance activities is not neglected in current research activities.

Human knowledge based analyses are frequently adopted in the primary studies (10/47). Such types of analyses are conceived to identify ATDIs in systems by extracting human knowledge through structured- or unstructured processes such as surveys, interviews, and questionnaires. We can conjecture that the reason of the high adoption of human knowledge based analyses is twofold: (i) some types of ATDIs need insights that are not documented (e.g., rationale behind a design decision, P11), and (ii) human knowledge provides validation to complement (semi-)automated ATD identification analyses (e.g., gathering feedback by presenting to stakeholders a visualization of the automatically identified ATDIs, P21).

Analyses based on *compliance checking* assess the discrepancy between intended and implemented architecture, and the deterioration of the architecture in time [85]. In most of the cases such approaches are tool supported (P3, P7, P43, P47). One of the studies (P3) relies on an uncommon technique to discover architectural compliance in software systems, namely a genetic algorithm.

Change impact analysis is about the evaluation of different design alternatives and/or the calculation of the effort required to resolve or avoid ATD in the future [86]. In many cases this typology of studies involves also cost analyses (P10, P21, P22). Approaches considering different alternative results are not common in our dataset (P9, P10, P11, P18, P21, P22). We can conjecture that this is due to the fact that only few papers consider ATD resolution, as reported in Section 2.5.6.

Few studies report analyses based on the *object-oriented (OO)* paradigm. In most of the cases

such approaches adopt consolidated software metrics and transfer the gathered results to the architectural level. For example, in P26 architecturally relevant classes are identified through code smells and the change history of the classes, while in P44 architecture quality is assessed by analyzing code smells among related OO classes. In general, the presence of such studies indicates that a portion of ATD analyses is rooted in code analysis, highlighting the thin line that separates software architecture and the source code of the system in this research area.

Only few studies focus on ATD *visualization* techniques. This observation is confirmed by Alves et al. [87] in their secondary study on TD. This result shows that this research area is only marginally considered by current research activities and requires to be further explored, especially considering its potential efficacy in the communication of ATD issues.

A minority of analyses involve a *manual classification* processes. For example P40 presents a visualization approach to highlight ATD-prone dependencies, based on which architects can manually select the most significant to be considered for refactoring. We can conjecture that the low presence of such type of analysis is due to the high effort required to carry out manual processes and their relative low scalability, which makes difficult their application to large software systems.

Only two studies focusing on *self-admitted ATD* were found in the primary studies. Such studies rely on code and commit comments where developers "self-admittedly" identify ATDIs. Such technique is commonly used to identify code related TD [88]. The low number of studies focusing on self-admitted ATD might indicate that it is still emerging and under-explored, or simply that architectural aspects of software systems are not frequently discussed in repositories (e.g. in commit messages, issue trackers).

2.5.4 Analysis Input

Our analysis identified five categories of ATD analysis inputs, namely: *Source Code*, *Evolutionary Data*, *Architectural Documentation*, *Survey* and *Issue Tracker*. As shown in Table 2.6, *Source Code* is the most recurrent analysis input (32/47). From this result we can observe that the majority of ATD identification techniques relies to a certain extent on the inspection of code-related properties. A smaller number of studies requires *Evolutionary Data* (16/47), such as commit history or measurements taken over time. Interestingly, two papers that utilize evolutionary data do not require source code as additional source, but rely on data extracted from human knowledge (P6), or use pre-existing architectural documentation (P10).

A lower number of studies takes *Architectural Documentation* as input (11/47). We conjecture that this is due to the difficultly to get access to industrial documentation, which is most often out of date, when available [89]. In line with the results reported in Section 2.5.3, 10 studies report approaches that require human knowledge as input source. Finally, a minority of studies takes *Issue Trackers* as additional input to source code (7/47). This latter typology of

Table 2.6 – Analysis input source

Input source	#Studies	Studies
Source Code	32	P1, P2, P3, P4, P5, P7, P9, P14, P15, P16, P17, P19, P20, P24, P25, P26, P28, P30, P32, P33, P34, P35, P37, P38, P39, P40, P41, P42, P43, P44, P45, P46, P47
Evolutionary Data	16	P1, P4, P5, P6, P10, P15, P16, P19, P20, P25, P26, P32, P34, P38, P42, P45
Architectural Documentation	11	P7, P10, P11, P13, P18, P22, P23, P24, P27, P31, P36
Human knowledge	10	P6, P8, P11, P12, P13, P18, P21, P23, P29, P36
Issue Tracker	7	P1, P15, P19, P20, P25, P32, P42

studies associate ATD to bug-related issues in source code. For example, in P20 the number of architectural flaws in a file is correlated to number of bugs in it, its change frequency, and the total amount of effort spent on it.

2.5.5 Temporal Dimension

Table 2.7 provides an overview of the papers taking into account the temporal dimension. In order to identify ATD, almost half of the papers considers the evolution of software systems through time (22/47). Here we can observe a discrepancy between the studies considering software evolution and the ones adopting *Evolutionary Data* as input (see Table 2.6). This difference is due to the studies relying on human knowledge extraction. In fact, those studies rely on human knowledge instead of evolutionary software artifacts in order to reason about the evolution of software systems.

Table 2.7 – Temporal dimension

Temporal dimension	#Studies	Studies
Not considered	25	P2, P3, P7, P8, P9, P11, P14, P16, P17, P21, P22, P24, P28, P30, P31, P33, P35, P37, P39, P40, P41, P43, P44, P46, P47
Considered	22	P1, P4, P5, P6, P10, P12, P13, P15, P18, P19, P20, P23, P25, P26, P27, P29, P32, P34, P36, P38, P42, P45

Surprisingly, from the gathered data we can observe that the majority of the studies does not consider the temporal dimension. Nevertheless we have to remark that this aspect is analysis-specific, and hence not required *per se* in order to identify ATD in a system. For example, P2

considers self-admitted TD in source comments of a single software release, and hence can pinpoint ATDI in source code without inspecting the entire version history of a system.

2.5.6 ATD Resolution

ATD resolution refers to refactoring strategies aimed to partially or completely remove identified ATDIs. Table 2.8 gives an overview of which studies consider ATD resolution. From the results we can observe that only a limited number of studies reports ATD resolution strategies (15/47). This could be a symptom of the relative young age of the ATD research field, where most of the research effort is still devoted to processes aimed at understanding the ATD phenomenon, rather than at resolving the identified ATD.

Table 2.8 – ATD resolution

ATD resolution	#Studies	Studies
Not considered	32	P42, P22, P41, P48, P24, P47, P2, P21, P31, P44, P45, P3, P14, P16, P30, P36, P38, P8, P40, P46, P27, P17, P5, P18, P43, P19, P35, P15, P20, P26, P33, P29
Considered	15	P11, P7, P28, P34, P9, P25, P10, P13, P1, P23, P37, P39, P6, P12, P4

2.5.7 Tool Support

Finally, from the studies we extracted which tool were utilized in order to carry out the ATDI identification processes. This attribute is meaningful both for (i) researchers who want to conceive new tool-based ATD identification techniques and (ii) practitioners needing tools to get further insights in their projects. A comprehensive list of the most recurrent tools is reported in Table 2.9.

Titan results to be the most used tool in our primary studies. Conceived by Xiao et al. [90], Titan introduces a new architecture model referred to as “design rule space”, intended to capture both the architecture and the evolutionary structure of systems in order to identify architectural issues. The second most used tools are Structure101[4] and SonarGraph[5]. These two commercial tools are static analyzers implementing functionalities to support software architects through dependency and modularity analysis at various levels of abstraction.

[4] http://structure101.com
[5] http://www.hello2morrow.com/products/sonargraph

Table 2.9 – Tool supported

Tool supported	#Studies	Studies
Titan	6	P1, P15, P19, P20, P33, P43
Structure101	4	P7, P28, P36, P47
SonarGraph	4	P7, P36, P38, P48
Understand	3	P25, P32, P48
inFusion	3	P28, P36, P38
SonarQube	3	P28, P36, P38
Arcan	2	P9, P14
CAST	2	P17, P36
ARAMIS	1	P44
CLIO	1	P46
Call Graph Extractor	1	P34
HUSACCT	1	P3
Hotspot Detector	1	P19
Lattix	1	P10
ModularityCalculator	1	P5
CommitAnalyer	1	P5
Ref-Finder	1	P26
Organic	1	P26
JSpIRIT	1	P26
SA4J	1	P28
iPlasma	1	P39

In total, a considerable number of primary studies (24/47) presents ATD identification approaches that require tool support. Nevertheless, this result has to be evaluated with caution. In fact, as further detailed in Section 2.6.1, only a subset of ATD identification techniques feature a publicly-available tool.

Main findings (RQ2). ATD identification is strongly rooted into TD techniques working at the source code level (this is evident when considering the abstraction level, input, and ATDI definitions of the proposed techniques).
Different interpretations of software architecture and ATDIs are proliferating in the state-of-the-art.
The literature proposes a large and heterogenous set of analysis types, ranging from the identification of architectural antipatterns, to dependency analysis, change impact analysis, and even manual classification of software artifacts.
ATD resolution is considered in less than one-third of the primary studies, indicating a promising research direction for the future. Similarly, the temporal dimension of ATD identification has been considered only in less than half of the primary studies.
A large number of tools for ATD identification are being proposed and used, but only a small portion of them is publicly available.

2.6 Results - Potential for Industrial adoption (RQ1.3)

In this section we report on the results regarding the potential of the studies for industrial adoption.

2.6.1 Tool Availability

The availability of a tool that implements a proposed ATD identification approach is required in order to enable the efficient adoption of such approach in industry.

Table 2.10 – Tool availability

Tool availability	#Studies	Studies
Not available	35	P1, P2, P4, P6, P8, P10, P11, P12, P13, P15, P16, P17, P18, P19, P20, P21, P22, P23, P24, P25, P27, P29, P30, P31, P32, P33, P34, P36, P39, P40, P41, P42, P43, P44, P47
Available	12	P3, P5, P7, P9, P14, P26, P28, P35, P37, P38, P45, P46

As shown in Table 2.10, only a small number of studies is implemented in an available tool. In particular, most of such studies makes use of a novel tool created *ad-hoc* for the ATD identification described in the paper or integrates tools that were already developed by the authors. For example, in P5 two publicly available tools developed by the authors are utilized: one to calculate modularity metrics (ModularityCalculator) and the other to calculate the average number of modified components per commit (CommitAnalyzer).

From these results we evince that, while some approaches are available in the form of a tool, this is not true for most of the studies (35/47). This might indicate that (i) numerous researches provide theoretical results and proof of concepts, and/or (ii) more effort is needed to ease the application of ATD identification approaches.

2.6.2 Industry Involvement

In order to assess the involvement of industry in the research related to ATD identification, we categorize the primary studies into three partially overlapping categories, namely: *academic*, *industrial* and *mixed*. A study is classified as *academic* if all authors are affiliated to universities or research institutes, *industrial* if all the authors are affiliated to industrial companies and, *mixed* if co-authors are from both academia and industry. The distribution of the primary studies according to this classification is reported in Figure 2.3.

As shown in Figure 2.3 the majority of the researches were conducted from an academic-only perspective (38/47), some studies emerged from a mixed perspective (7/47) and industry-

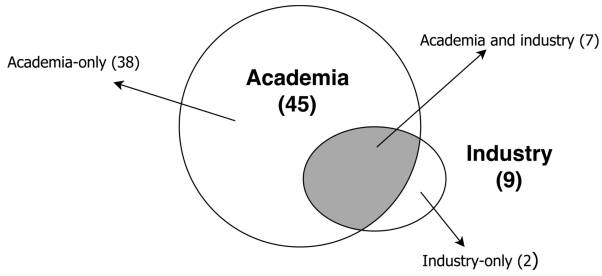


Figure 2.3 – Distribution of industry involvement

only studies are rare (2/47). From these results we can observe that the research topic is still addressed mostly from an academic perspective; more partnerships between industry and academia are necessary in order to enable beneficial knowledge exchanges and acquire a more comprehensive understanding of the problems and applicable solutions.

2.6.3 Rigor and Industrial Relevance

To gain further insights into the potential for industrial adoption of the ATDI identification approaches, we evaluated the *rigor* and *industrial relevance* of the studies. This process was carried out by applying the well-defined classification model introduced by Ivarsson et al. [81]. Accordingly, *Rigor* refers to the accuracy or exactness of the research method used, and is subdivided into three categories: *Context*, *Study design* and *Validity*. These categories assume values "0", "0.5", and "1" reflecting the quality of their description. *Industrial relevance*, instead, considers experimental *Subjects*, *Context*, *Scale*, and *Method*, which assume values "0" or "1".

2.6.3.1 Rigor

Figure 2.4 shows the distribution of the primary studies among the three rigor categories. We observe that the *context* considered in the studies is in most of the cases reported but described schematically (24/47). This indicates that potential impediments could be encountered when contexts different from the ones reported in the studies are considered. The *study design* is generally characterized by a medium or strong description (19/47 and 23/47, respectively), enabling the reader to clearly understand the variables considered, the treatments adopted, etc. Concerning *validity*, a relatively low number of studies discusses threats to validity (10/47). In most of the studies validity is only mentioned (19/47) or fully neglected (18/47). This suggests that more effort should be spent in documenting the validity of the researched approaches,

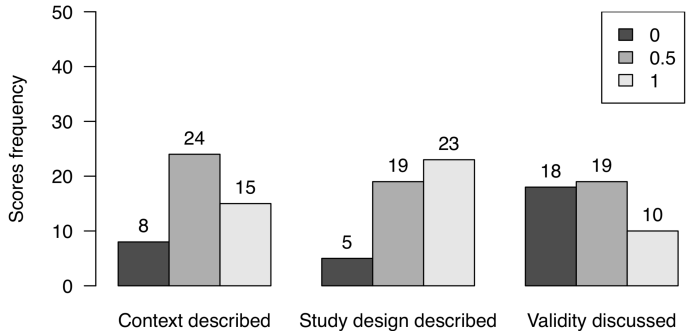


Figure 2.4 – Rigor of primary studies

necessary to increase their potential for industrial adoption.

2.6.3.2 Industrial relevance

As shown in Figure 2.5, most of the studies consider representative industrial *subjects* (33/47) and utilize evaluation *methods* that could be efficiently applied in an industrial setting (37/47). From the data we can observe that, the fact that ATD is a problem rooted in practice is not neglected by researchers. In fact, the studies present approaches that can be easily applied in practice as well as industrial case studies utilized for their evaluation. Naturally, the size of the systems considered is in the majority of the cases of industrial *scale*, too (35/47).
On a less positive note, the *context* is not always representative of the intended usage of the researched approach, as evaluations are often preformed in an academic setting instead of an industrial one (19/47). This is reflected also in the high number of academic-only authors reported in Section 2.6.2.

2.6.3.3 Combined analysis of rigor and industrial relevance

By jointly considering the *rigor* and *industrial relevance* distributions of primary studies, we can sketch which future steps should be taken in order to increase the potential for industrial adoption of the research results in ATD identification. As illustrated in Figure 2.6, the practical roots of ATD research seem to deeply influence the *industrial relevance* of the primary studies, the vast majority of which has high cumulative scores for such attribute. A higher variability and lower scores can instead be noticed if *rigor* is considered. We can hence conclude that in future research more effort should be put into rigorously describing the *context* and the *threats*

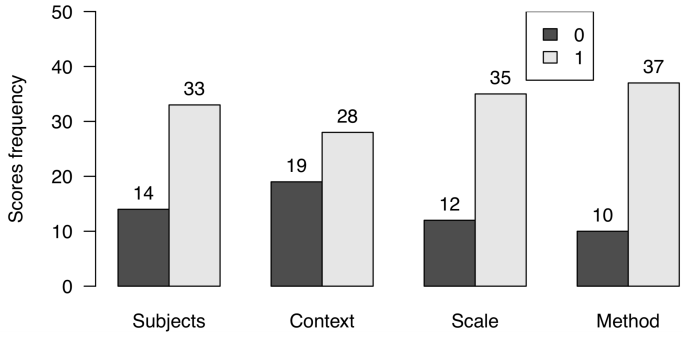


Figure 2.5 – Industrial relevance of primary studies

to validity, in order to increase rigor quality.

Main findings (RQ3). The lack of available tool support for the majority of the proposed ATD identification approaches hinders knowledge transfer and industrial adoption.
To date, most research is academic-only; to further the field more collaboration between academia and industry would accelerate knowledge transfer and tuning the research focus on the most-critical industrial problems.
Research rigor (in terms of reusable study designs) and industrial relevance (in terms of targeted industrial subjects and scale, and used methods) are potentially ready for industrial adoption. However, the limitations of the majority of the primary studies (in terms of context description and discussion of the validity threats) represent a potential risk to their successful industrial application.

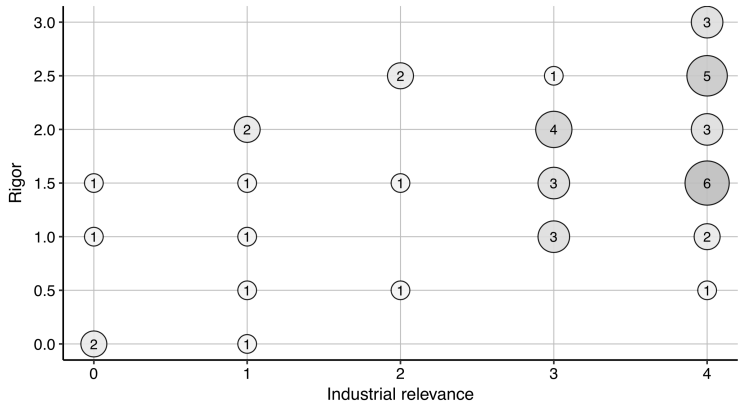


Figure 2.6 – Rigor and relevance of primary studies

2.7 Threats to validity

In this section we discuss the threats to validity of our research. In general, in order to ensure the high quality of the data extracted, a well-defined research protocol was established before carrying out the data collection. In addition, all research activities were designed and carried out by adhering to a set of well-accepted guidelines for systematic mapping studies [80, 74, 73]. By formalizing such guidelines we established the protocol that was strictly followed throughout the study, as documented in Section 6.3.1 and further detailed in the provided replication package. In addition, in order to lower potential sources of bias, crucial considerations that emerged during the research were discussed jointly by all the authors. Despite adhering to a systematic literature review approach, potential threats to validity are still unavoidable, as discussed in the remainder of this section along with how we mitigated them.

External validity. The most significant threat to external validity consists of the potentially low representativeness of the primary studies. In order to mitigate this threat, we adopted for the identification of potentially relevant studies the most encompassing digital library[6] (*Google Scholar*) and search query. Another threat to external validity is the adoption of a specific set of query keywords. To mitigate this threat the results of the automated search query were further extended by executing a backward and forward snowballing process. In order to have control over the quality of the primary studies, we exclusively considered peer-reviewed papers, and hence excluded "grey literature", e.g. white papers, editorials, etc. We deem that this does not

[6]Selected after a preliminary execution of the search query on: *Google Scholar*, *Scopus*, *IEEE Explore*, *ACM Digital Library*, and *Web of Science*.

constitute an additional threat, as peer-review processes are a standard requirement of high quality publications. Finally, we utilized a set of well-defined inclusion and exclusion criteria, which rigorously guided our manual selection of the literature.

Internal validity. To mitigate potential threats to internal validity, we established *a priori* a rigorous research protocol that guided all the research activities. In addition, the classification framework utilized was defined iteratively by strictly adhering to the keywording process [80]. For the synthesis of the collected data, simple and well-established descriptive statistics were adopted. In addition, sanity tests on the extracted data were used by cross-analyzing different parameters of the established classification framework.

Construct validity. In order to ensure that the primary studies were suited to answer our research questions, we manually carried out the selection of primary studies according to a predefined set of well-documented inclusion and exclusion criteria. The results of such process were further expanded by conducting an additional iterative backward and forward snowballing process. In addition, as advised by Wholin et al. [73], a random sample of 10 studies were selected and analyzed independently by all 3 researchers in order to guarantee the alignment of the analyses.

Conclusion validity. Potential sources of bias resulting from the data extraction and analysis processes were mitigated by strictly adhering to an *a priori* defined protocol, explicitly conceived to gather the data required to answer our research questions. To further mitigate threats to conclusion validity, best practices from several well known guidelines for systematic literature reviews [80, 74, 73] were followed. These guidelines were strictly adhered to throughout the entirety of our research activities, and are thoroughly documented to ensure that our research approach is transparent and replicable.

2.8 Conclusions

This chapter presents a systematic mapping study on ATD identification, the first and foremost building block of ATD management. Starting from 509 potentially relevant studies, we rigorously analyzed 47 primary studies via a classification framework dedicated to ATD identification. Our analysis provides a characterization for ATD identification techniques in terms of publication trends, their characteristics, and their potential for industrial adoption.

Furthermore, our analysis unveils a series of promising trajectories for future research on ATD, such as (i) the possibility leverage existing source-code analysis tools to conduct ATD-specific analyses, (ii) the current need to further involve industrial parties when studying ATD identification phenomena, and (iii) a lack of technology-specific ATD identification approaches. We explore the research opportunities emerging from this first research step in the following chapters. Specifically, in Chapter 3, we present an ATD index based on architectural design rule violations. The need to further involve practitioners in academic research is tackled in

Chapter 4, where we present a theory of ATD grounded in the knowledge of technical leaders. The possibility to conceive technology-specific analysis approaches is presented in Part II of this thesis, were we concentrate on ATD specific to the Android ecosystem, and present a methodology targeting the identify of ATD in such context.

3 ATDx: An Architectural Technical Debt Index

Inside every large problem is a small problem struggling to get out.

Charles Hoare

This chapter is based on:
R. Verdecchia, P. Lago, I. Malavolta, and I. Ozkaya, *ATDx: Building an Architectural Technical Debt Index*, International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE), 2020 [54].
R. Verdecchia, P. Lago, I. Malavolta, and I. Ozkaya, *Empirical Evaluation of an Architectural Technical Debt Index in the Context of the Apache and ONAP Ecosystems*, Under Journal Submission [55].

Chapter 3. ATDx: An Architectural Technical Debt Index

This chapter answers the fourth research question of this thesis (RQ4). Specifically, it documents the preliminary version of an index (ATDx) conceived in order to provide an eagle-eye overview of the health of software systems from the point of view of code related architectural technical debt. The index is established through a large-scale, empirically based, analysis of design issues which can be statically detected through the adoption of a prominent software quality tool, namely SonarQube. By leveraging techniques such as repository mining, statistical analysis, and a blend of other quantitative and qualitative approaches, we present in this chapter portions of our ongoing research effort to expand our knowledge on the detection and management of architectural technical debt.

Contents

3.1 Introduction

Architectural Technical Debt (ATD) in a software-intensive system denotes architectural design choices which, while being suitable or even optimal when adopted, lower the maintainability and evolvability of the system in the long term, hindering future development activities [83]. With respect to other types of debt, *e.g.,* test debt [91] or build debt [29], ATD is characterized by being widespread throughout entire code-bases, mostly invisible to software developers, and of high remediation costs [92].

Due to its impact on software development practices, and its high industrial relevance, ATD is attracting a growing interest within the scientific community and software analysis tool vendors [1]. Notably, over the years, numerous approaches have been proposed to detect, mostly via source code analysis, ATD instances present in software intensive-systems. Such methods rely on the analysis of symptoms through which ATD manifests itself, and are conceived to detect specific types of ATD by adopting heterogeneous strategies, ranging from the inspection of bug-prone components [93], to the analysis of dependency anti-patterns [94], and the evaluation of components modularity [95]. Additionally, numerous static analysis tools, such as NDepend[1], CAST[2], and SonarQube[3], are currently available on the market, enabling to keep track of such symptoms of technical debt and architecture-related issues present in code-bases. These existing academic and industrial approaches focus on fine-grained analysis techniques, considering *ad-hoc* definitions of technical debt and software architecture, in order to best fit their analysis processes to technical debt assessment. Nevertheless, to date, how to gain an informative and encompassing viewpoint of the (potentially highly heterogeneous [57]) ATD present in a software-intensive system independent of the tools at hand is still an open question.

In order to fill this gap, in this study we present a refined version of ATDx [54], an approach designed to provide a data-driven, intuitive, and actionable insights on the ATD present in a software-intensive system. ATDx consists of a theoretical, multi-step, and semi-automated process, concisely entailing (i) the reuse of architectural rules supported by third-party analysis tools, (ii) calculation of architectural rule violations severity, based on the comparison of normalized values across a software portfolio, and (iii) the aggregation of analysis results into a set of customizable ATD dimensions. Similarly to other studies in the literature (*e.g.,* [96]), in this paper we refer to *software portfolio* as the set of software projects (also referred to as *software assets*) owned by a single company.

ATDx is designed to serve two types of stakeholders: (i) *researchers* conducting quantitative studies on source-code related ATD and (ii) *practitioners* carrying out software portfolio analysis and management, to suitably detect ATD items and get actionable insights about the ATD

[1] https://www.ndepend.com
[2] https://www.castsoftware.com/products/code-analysis-tools
[3] https://www.sonarqube.org

present in their systems according to their organizational and technical needs.

This study builds upon the research in which ATDx was preliminarily reported [54] by (i) refining the ATDx in order to address some of its drawbacks (see Section 3.2), and (ii) conducting an empirical evaluation of the approach.

We carry out an empirical evaluation of the ATDx approach involving two open-source software ecosystems (Apache and ONAP), 237 software projects, and 233 open-source software contributors. The gathered results shed light on the representativeness and actionability of ATDx, and provide further insights of the benefits and drawbacks which characterize the approach. Among other, the most relevant characteristics of ATDx are: (i) analysis tool and programming language independence, (ii) data-driven results, rather than based on *a priori* defined severities, remediation costs, and metric thresholds (iii) extensibility, and (iv) customizability to specific application domains and portfolios.

The main contributions of this paper are the following:

- the **evolution of ATDx**, an approach providing a multi-level index of architectural technical debt; refined by replacing the outlier-based calculation of the original approach [54] with a severity clustering algorithm;
- a detailed description of the **process for building an instance of ATDx**, supporting the independent implementation of an instance of ATDx by researchers and practitioners;
- an **empirical evaluation** of the representativeness and actionability of the ATDx approach based on SonarQube , involving two software ecosystems, 237 software projects, and 233 software contributors, supported by the complete replication package[4], and a thorough discussion of the uncovered ATDx advantages and drawbacks.

The remainder of the paper is structured as follows. In the next section, we present the theoretical framework underlying the ATDx approach, followed by the formalization of the approach, and the description of the steps required to implement an instance of ATDx. In Sections 3.3 and 3.4 we document the planning and execution of the empirical evaluation, respectively. The results of the empirical evaluation are then reported in Section 6.3.4. In Section 3.6 the discussion of the results is reported, while in Section 6.3.5 we elicit the potential threats to validity which may have influenced our results. In Section 4.4 we present and discuss the related work. Finally, Section 3.9 draws conclusions and hints at future work.

[4]https://github.com/S2-group/ATDx_replication_package

3.2 The ATDx Approach

In this section, we provide the definitions of attributes on which the calculation of ATDx relies (Section 3.2.1), the ATDx formalization (Section 3.2.2), and describe the steps for building ATDx (Section 3.2.3).

3.2.1 Definitions

Definition 1. Architectural rule. Given a source code analysis tool T and the set of its supported analysis rules R^T, the *architectural rules* AR^T supported by T are defined as the subset of all rules $R_i^T \in R^T, i = \{1,\ldots,|R^T|\}$ such that:

- R_i^T is relevant from an architectural perspective, *i.e.,* strongly influences one choice of structures for the architecture [97];
- R_i^T is able to detect a technical debt item, *i.e.,* "design or implementation constructs that are expedient in the short term but that set up a technical context that can make a future change more costly or impossible" [83].

In ATDx, we consider every AR_i^T as a function $AR_i^T : E \rightarrow \{0,1\}$, where E is the set of architectural elements according to a granularity level (see below). In case that an element $e \in E$ violates rule AR_i^T, then $AR_i^T(e)$ returns 1, and 0 otherwise.

For example, a rule AR_i^T checking that method overrides should not change contracts is (i) *architectural* since it predicates on the high-level structure of a Java-based software project (*i.e.,* its inheritance tree), and (ii) *related to technical debt* as violating such rule might not lead to immediate repercussions, but could potentially cause unexpected behaviour and cumbersome refactoring as the software project evolves.

Definition 2. Architectural Rule Granularity level (Granularity level). Given an architectural rule AR_i^T, its granularity level Gr_i^T is defined as the smallest unit of the software project being analysed which may violate AR_i^T, *e.g.,* a class, a method, or a line of code. As an example, if we consider a rule which deals with cloned classes, its corresponding granularity level is "class". Such mapping of architectural rules to different granularity levels enables us to evaluate and compare the occurrence of rules violations across different software projects at a refined level of precision, instead of trivially adopting a single metric for the size of software projects for all the rules in AR^T, *e.g.,* source lines of code (SLOC). In addition, it enables us to assess the scope of the technical debt and as needed differentiate from defects.

Definition 3. ATD Dimension. Given a set of architectural rules AR^T for an analysis tool T,

the set of ATD dimensions ATDD^T contains subsets of architectural rules $AR_i^T \subseteq AR^T$ with similar focus. One architectural rule AR_i^T can belong to one or more ATD dimensions $\text{ATDD}_j^T \subseteq \text{ATDD}^T$ and the mapping between AR_i^T and ATDD_j^T is established by generalizing the semantic focus of AR_i^T. For example, if an architectural rule AR_i^T deals with the conversion of Java classes into Java interfaces, the AR_i^T could fall under the general *Interface* ATD dimension.

In ATDx, we use the 3-tuple $\langle AR_i^T, \text{ATDD}_j^T, Gr_i^T \rangle$ to represent the mapping of each architectural rule AR_i^T to its granularity level Gr_i^T and ATD dimensions ATDD_j^T. It is important to note that, while an AR_i^T can be associated to one and only one granularity level Gr_i^T, an AR_i^T can be mapped to multiple dimensions ATDD_j^Ts, and vice versa.

3.2.2 ATDx Formalization

ATDx aims to provide a birds-eye view of the ATD present in a software project by analyzing the set of architectural rules AR^T supported by an analysis tool T, and subsequently aggregating the analysis results into different ATD dimensions ATDD^T.

The goal of ATDx is portfolio analysis of projects in respect to their level of ATD. Intuitively, starting from a dataset of AR^T and Gr^T values belonging to a set of software projects S. ATDx performs a statistical analysis on the elements contained in the dataset, in order to classify the severity of the architectural rule violations of the software projects. The level of severity the SUA exhibits for each rule $AR_i^T \in AR^T$, is then reported as a constituent part of the ATD dimension $\text{ATDD}_i^T \in \text{ATDD}^T$ mapped to AR_i^T. Notice that the ATDx analysis results of a specific system-under-analysis (SUA) are *relative* to the other projects S in the same portfolio, and hence should not be interpreted as absolute values.

ATDx is based on the calculation of the number of architectural rule violations of a software project S (normalized over the size of S) in order to compare the occurrence of rule violations across projects of different sizes. Specifically, for each architectural rule AR_i^T, we first calculate $AR_i^T(\text{S})$, defined as the set of architectural elements in S violating AR_i^T, *i.e.*,

$$AR_i^T(\text{S}) = \bigcup_{e \in Gr_i^T(\text{S})} ar_i^T(e) \tag{3.1}$$

where $Gr_i^T(\text{S})$ is the set of all elements e in S according to the granularity Gr_i^T (*e.g.,* the set of all Java classes in a Java-based project), and $ar_i^T(e)$ is a function returning e if the element e violates the architectural rule AR_i^T, the empty set otherwise.

Subsequently, we calculate $\text{NORM}_i^T(\text{S})$, defined as the normalized number of architectural

rule violations $|AR_i^T(\mathrm{S})|$ over the total number of elements e according to granularity Gr_i^T, *i.e.,*

$$\mathrm{NORM}_i^T(\mathrm{S}) = \frac{|AR_i^T(\mathrm{S})|}{|Gr_i^T(\mathrm{S})|} \tag{3.2}$$

where $|Gr_i^T(\mathrm{S})|$ is the size of S expressed according to granularity level Gr_i^T (*e.g.,* the total number of Java classes in S), and $|AR_i^T(\mathrm{S})|$ is the total number of violations of rule R_i^T (see Formula 3.1).

Once the $\mathrm{NORM}_i^T(\mathrm{S})$ for rule AR_i^T in S is calculated, we statistically establish its *severity.* In order to do so, we require the set NORM_i^T, which contains the values of $\mathrm{NORM}_i^T(\mathrm{S})$ for each software project belonging to the portfolio, *i.e.,*

$$\mathrm{NORM}_i^T = \{\mathrm{NORM}_i^T(\mathrm{S}_1), \ldots, \mathrm{NORM}_i^T(\mathrm{S}_n)\} \tag{3.3}$$

where n is the total number of projects belonging to the considered portfolio of software projects.

Given the calculation of NORM_i^T, we can establish the severity of the $\mathrm{NORM}_i^T(\mathrm{S})$ measurement by comparing its value with the other ones contained in NORM_i^T. More specifically, given the set of values NORM_i^T and the value of $\mathrm{NORM}_i^T(\mathrm{S})$, we define the function *severity* as:

$$severity : X^n \times [0,1] \rightarrow \{0,1,2,3,4,5\} \tag{3.4}$$

where $X = [0,1]$ and n is the total number of software projects belonging to the portfolio. The *severity* function returns a discrete value between 0 and 5, indicating the level of severity of $\mathrm{NORM}_i^T(\mathrm{S})$ w.r.t. the other values in NORM_i^T. In order to do so, we adopt a clustering algorithm, namely CkMeans [98], which guarantees optimal, efficient, and reproducible clustering of univariate data (*i.e.,* in our case, NORM_i^T values). Consequently, this step consists of identifying the severity cluster of $\mathrm{NORM}_i^T(\mathrm{S})$ that contains similar NORM_i^T values of other software projects within the portfolio. The usage of the CkMeans algorithm replaces the outlier-based calculation of ATD on which the original ATDx approach was based [54]; this decision allows us to gain finer-grained results (*i.e.,* a discrete value between 0 and 5 instead of a boolean value).

An example of AR^T values distribution, and relative severity clustering, is provided in Figures 3.1 and 3.2. As we can observe in Figure 3.1, the majority of the projects possess NORM_i^T values between 0.0 and 0.1, which are grouped via CkMeans into three distinct clusters, as depicted in Figure 3.2. Such clusters correspond to the lowest levels of severity, namely severity 0, 1, and 2 respectively. The other three clusters, possessing centers (*i.e.,* weighted mean of cluster) of respectively 0.12, 0.21, and 0.5, correspond to the higher levels of severity, namely severity levels 3, 4, and 5. From the clustering depicted in Figure 3.2 we see that, according to their distribution, most projects are classified as possessing low severity (severity <= 2), while only a smaller number of projects possesses a relatively high severity (severity >= 3).

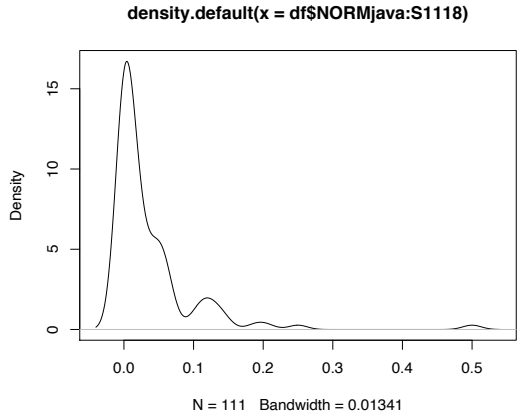


Figure 3.1 – Example of kernel density plot representing a NORM_i^T distribution

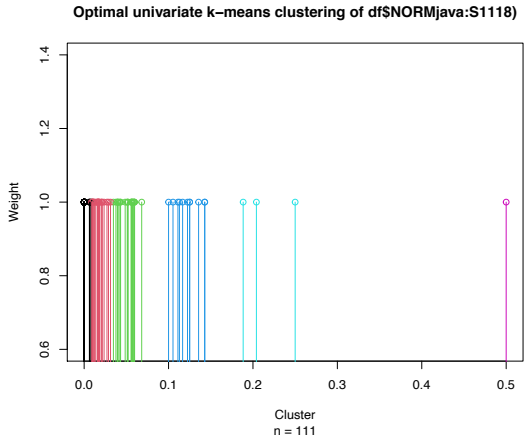


Figure 3.2 – Example of severity calculation via CKMeans clustering, where different colours indicate different clusters

In order to provide an overview of the ATD dimensions of a software project S, for each ATD dimension $\text{ATDD}_j^T \subseteq \text{ATDD}^T$ we define the value of $\text{ATDD}_j^T(\text{S})$ as the average severity of the AR_i^T mapped to it, *i.e.,*

$$\text{ATDD}_j^T(\text{S}) = \frac{\sum_{i=1}^{n} severity(\text{NORM}_i^T, \text{NORM}_i^T(\text{S}))}{n} \tag{3.5}$$

where n is the total number of rules in AR^T mapped to ATDD_j^T.

Finally, we define an overall value $\text{ATDx}^T(\text{S})$, embodying the overall architectural technical debt of S calculated via our approach, as the average value of all the defined ATD dimensions ATDD^T, *i.e.,*

$$\text{ATDx}^T(\text{S}) = \frac{\sum_{j=1}^{n} \text{ATDD}_j^T}{n} \tag{3.6}$$

where n is the total number of ATD dimensions ATDD^T considered in the specific implementation of ATDx.

3.2.3 ATDx Building Steps

In this section, we report the steps for building ATDx. It is important to note that the whole process is generic, *i.e.,* it is not bound to any specific analysis tool or technology and extensible. The described process can be performed by both (i) researchers investigating ATD phenomena and (ii) practitioners analyzing their own software portfolios. In fact, following the steps of the process allows its users to implement the instance of ATDx which best fits their specific technical, organizational, and tool-related context.

Figure 6.1 presents the building steps for implementing the ATDx approach. Given an analysis tool T (*e.g.,* SonarQube), five steps are required to build an instance of ATDx, namely: (i) the identification of the set of architectural rules belonging to AR^T, (ii) the formulation of the 3-tuples in the form $\langle AR_i^T, Gr_i^T, ATDD_j^T \rangle$, (iii) the execution of T on a set of already available software projects to form the dataset of $AR_i^T(S)$ measurements, (iv) the execution of the ATDx analysis on the constructed dataset, and (v) the application of the ATDx approach on the specific SUA.

Figure 3.3 – Overview of the ATDx building steps [54]

3.2.3.1 Step 1: Identification of the AR^T set

The first step of the ATDx building process is the identification of a set of architectural rules AR^T that will be used as input to the subsequent steps of the process. Specifically, given an analysis tool T and its supported analysis rules R^T, a manual inspection is carried out in order to assess which of its rules qualify as AR^T according to the criteria presented in Definition 1. This process can be carried out either by inspecting the concrete implementation of the rules R^T under scrutiny, or by consulting the documentation of T, if available.

3.2.3.2 Step 2: Formulation of 3-tuples $\langle \mathbf{AR}_i^T, \mathbf{ATDD}_j^T, \mathbf{Gr}_i^T \rangle$

After the identification of AR^T, the 3-tuples $\langle AR_i, \text{ATDD}_j, Gr_i \rangle$ are established by mapping each rule AR_i^T to (i) one or more architectural technical debt dimensions $ATDD_j^T$ and (ii) the granularity level GR_i^T of the rule.

The process of mapping an AR_i^T to its corresponding architectural dimensions ATDD_j^T is conducted by performing iterative *content analysis* sessions with open coding [99] targeting the implementation or documentation of the rule in order to extract the semantic meaning of the rule. More in details, once the semantic meaning of each rule is well understood, the AR_i^T under scrutiny is labeled with one or more keywords expressing schematically its semantic meaning. Such analysis is carried out in an iterative fashion, *i.e.,* by continuously comparing the potential $ATDD_j^T$ associated to the AR_i^T under analysis with already identified dimensions, in order to reach a uniform final $ATDD^T$ set.

The process of mapping an architectural rule AR_i^T to its corresponding level of granularity Gr_i^T is also carried out via manual analysis of the architectural rule, and subsequently identifying the unit of analysis that the rule considers (*e.g.,* function, class, or file level).[5]

It is important to note that Steps 1 and 2 are performed only once for the whole portfolio and, depending on the tool and its rules, their corresponding 3-tuples can be also reused across different portfolios.

[5] In the fortunate instance in which the GR_i^T mapped to AR_i^T is explicitly specified in the source from which the rules R^T are gathered, such information should be preferred over a manual inspection of the rule.

3.2.3.3 Step 3: Building the AR^T(SUA) dataset

After the identification of the AR^T set (Step 1), it is possible to build the dataset of $AR^T(S)$ measurements. This process consists of (i) identifying an initial set of projects to be considered for inclusion in the portfolio, (ii) carrying out a quality filtering process in order to filter out irrelevant projects (*e.g.,* demos, examples) and (iii) calculating the $AR^T(S)$ sets and extracting the $|Gr^T(S)|$ values of each project included in the portfolio. The selection of the initial portfolio of projects to be considered for inclusion is a design choice specific to the concrete instance of ATDx. In other words, such choice is dependent on the analysis goal for which ATDx is adopted, the availability of the software projects to be analyzed, and the tool T adopted to calculate the $AR^T(S)$ sets. It is important to bear in mind that, given the statistical nature of ATDx, having a low number of projects in this step would not lead to meaningful ATDx analysis results (as further discussed in Section 3.6).

As for the selection of the projects to be considered, the step of carrying out a quality-filtering process on the initial set of projects depends on the setting in which ATDx is implemented. In the case that ATDx is used for an academic study, *e.g.,* by considering open-source software (OSS) projects, this step must be carried out to ensure that no toy software-projects (like demos or software examples written for educational purposes) are included in the final software portfolio to be considered [100].

After the identification of a final set of projects to be considered for analysis, the $AR^T(S)$ sets are calculated for each software project S in the portfolio. The execution of such process varies according to the adopted analysis tool T. In addition, during this step also the cardinalities of the granularity dimensions $|Gr_i^T(\mathrm{S})|$ are for each project S in the portfolio. Such values will be used in the next ATDx steps.

3.2.3.4 Step 4: ATDx Analysis

Once the $AR^T(S)$ and $Gr^T(S)$ sets are calculated for the whole portfolio, the architectural technical debt of the projects can be assessed (see Section 3.2). Specifically, this step takes as input $AR^T(S)$ and $Gr^T(S)$ sets for all the projects in the portfolio, and outputs the ATDD$^T(S)$ and ATDx$^T(S)$ values of each project. It is worth noticing that this process is incremental. Indeed, after a first execution of the ATDx approach on the whole portfolio, it is possible to carry out further ATDx analyses on additional projects by relying on the previously formulated 3-tuples $\langle AR_i^T, \mathrm{ATDD}_j^T, Gr_i^T \rangle$ and the pre-calculated intermediate values of the ATDx analysis $NORM_i^T$.

3.2.3.5 Step 5: Applying ATDx to a SUA

After the execution of ATDx on all projects in portfolio, the resulting ATDD^T and ATDx ^{T} values of a specific SUA can be computed.

Algorithm 1: Computing the $ATDD^T$ dimensions and the $ATDx^T$ value for a single SUA

Input: SUA, AR^T, NORM^T, ATDD^T
Output: ATDD^T(SUA), $ATDx^T$(SUA)

```
1  dimensions ← empty dictionary
2  atdx ← 0
3  for all dimensions j in ATDD^T do
4      dimensions[j] ← 0
5  for all rules AR_i^T in AR^T do
6      violations ← AR_i^T(SUA)
7      normalizedViolations ← NORM_i^T(SUA)
8      dimensions[j] ← dimensions[j] + severity(NORM_i^T, normalizedViolations)
9  for all entries j in dimensions do
10     dimensions[j] ← dimensions[j] / getNumRules(j)
11     atdx ← atdx + dimensions[j]
12 atdx ← atdx / |ATDD^T|
13 return dimensions, adtx
```

As shown in Algorithm 1, the computation of ATDD^T and ATDx ^{T} takes as input 4 parameters: (i) the SUA, the set of rules AR^T, the NORM^T values computed in the step 4, and the set of dimensions ATDD^T defined in step 2. The outputs of the algorithm are two, namely: (i) the set of ATD values of the SUA, ATDD^T(SUA) (one for each dimension) and (ii) $ATDx^T$(SUA). The outputs of the algorithm serve two different purposes; Specifically, the ATDD^T values provide support in gaining more insights in the severity of the ATD according to the identified ATD dimensions, while the ATDx ^{T} value provides a *unified overview* of the ATD present in the SUA. After setting up the initial variables for containing the final output (lines 1-2), the algorithm builds a dictionary containing an entry for each dimension in ATDD^T, with the name of the dimension as key and 0 as value (lines 3-4). Then, the algorithm iterates over each rule in AR^T (line 5) and collects the number of its violations, both raw (line 6) and normalized by the level of granularity of the current rule (line 7). Then the entry of the `dimensions` dictionary corresponding to the dimension of the current rule is incremented by the severity level of NORM$_i^T$ as defined in Equation 3.4 (line 8). For each dimension j (line 9) we (i) average its current value within the total number of rules belonging to j in order to mitigate the potential effect that the number of rules

belonging to the dimension may have (line 10) and (ii) increment the current ATDx^T with the computed score (line 11). Finally, the ATDx^T value is normalized by the total number of dimensions supported by all AR^T rules (line 12) and both dimensions and ATDx values are returned (line 13).

ATDx in a Nutshell. ATDx is a data-driven approach providing an overview of code-related architectural technical debt of a software-intensive system. The approach, based on the analysis of a software portfolio, uses pre-computed architectural rule violations (AR^T) and granularity levels (Gr^T) to calculate the severity level of violations via a clustering algorithm. Results are aggregated into different architectural technical debt dimensions (ATDD^T).

3.3 Empirical Evaluation Planning

We conduct an *in vivo* empirical evaluation to assess the viability of ATDx. In the remainder of this section we report (i) the goal and research questions of the empirical evaluation (Section 3.3.1) and (ii) its design (Section 3.3.2).

3.3.1 Goal and Research Questions

Intuitively, with our empirical evaluation we aim to understand if the ATDx analysis results faithfully represent the ATD present in real life software projects. In addition, we aim to assess if the ATDx analysis results are actionable, *i.e.*, they motivate practitioners to refactor their ATD. More formally, we formulate the goal of our evaluation by following the Goal-Question-Metric template [101] as follows:

Analyze ATDx analysis results
For the purpose of evaluating their representativeness and their ability to stimulate action
With respect to architectural technical debt
From the viewpoint of software practitioners
In the context of open-source software projects

It is important to note that ATDx can be applied to both open-source and proprietary software. In this study we focus on open-source software projects due to (i) the

availability of rich data about their source code and development process and (ii) the ease of mining of such type of software projects with respect to proprietary ones [102]. Further considerations on the this empirical evaluation design decision are reported in Section 6.3.5.

By taking into account our research goal, we can derive the following two research questions:

RQ1: *To what extent are the ATDx results **representative** of the architectural technical debt present in a software project?*

By answering this research question, we aim at assessing the representation condition [103] of ATDx, *i.e.,* the extent to which the characteristics of the ATD present in a software intensive-system are preserved by the numerical relations calculated via the ATDx approach. In other words, this research question evaluates to which extent the ATDx analysis results are representative of the ATD in a software project, both by individually considering individually the results for each software-intensive system, than by comparing results across different systems.

RQ2: *To what extent do the ATDx results **stimulate action** of developers to address their architectural technical debt?*

By answering this research question, we aim to assess the extent to which the ATDx analysis results stimulate developers to address ATD, *i.e.,* if the results motivate developers to actively manage the ATD detected via ATDx.

3.3.2 Empirical Evaluation Design

The empirical evaluation is designed according to our research questions and includes all the building steps of the ATDx approach described in Section 3.2. The evaluation is composed of nine main phases:

- **Phase 0** – Identification of the analysis tool T to be used in the evaluation.
- **Phase 1** – Identification and classification of the set of architecturally-relevant rules (*i.e.,* AR^T); this step corresponds to the ATDx building Steps 1 and 2 in Figure 6.1.

- **Phase 2** – Identification of one or more software portfolios to be analyzed.
- **Phase 3** – Establishment of the AR^T dataset(s) for the selected software portfolio(s); this step corresponds to Step 3 in Figure 6.1.
- **Phase 4** – Analysis of the dataset(s) via ATDx; this step corresponds to Steps 4 and 5 in Figure 6.1.
- **Phase 5** – Identification of a curated set of contributors of the selected software portfolio(s).
- **Phase 6** – Generation of personalized ATDx reports.
- **Phase 7** – Distribution of the ATDx reports.
- **Phase 8** – Online survey on the ATDx analysis results.

In the remainder of this section we explain each phase of our empirical evaluation; we present the phases in general terms, so that independent researchers can fully reuse them in future replications of this study. Then, in Section 3.4 we provide the technical details about how we implemented and executed each phase in the context of (i) Java projects, (ii) the SonarQube analysis tool, and (iii) the Apache and ONAP software ecosystems.

In order to evaluate ATDx, we implement a concrete instance of ATDx by following the building steps presented in Section 3.2. As a first step, in **Phase 0** we select a source code analysis tool T, implementing the R^T rules.

Phase 1 aims at identifying a set of AR^T rules on which the ATDx approach will be based. Specifically, the AR^T identification process is conducted by considering: (i) the soundness of the AR^T rules, demonstrated by industrial adoption and scientific evidence, (ii) the industrial relevance of the tool implementing the AR^T rules, and (iii) the feasibility of calculating AR^T values. In addition, during this phase, the identified AR^T rules are manually classified, in order to derive the 3-tuples $\langle AR_i^T, \text{ATDD}_j^T, Gr_i^T \rangle$ required by the ATDx approach.

As discussed in Section 3.2, the ATDx calculation relies on a portfolio of software projects. Hence, in order to gather ATDx analysis results, in **phase 2** we identify the software portfolio(s) that will be used as experimental subject in our evaluation. The focus on software portfolios, rather than a collection of unrelated software projects, allows to focus on software projects that potentially share a similar context, and overlapping contributors, and hence are closer to the envisioned usage scenario of ATDx. Accordingly, in case of more than one portfolio is identified during this

phase, the portfolios will be analyzed via ATDx independently. Driving factors for the identification of software portfolios is the availability of the software projects contained in the portfolio, and the possibility to calculate AR^T values for the portfolio, according to the AR^T rule set identified in the previous empirical evaluation phase.

In **phase 3** we compute the values of AR^T and Gr^T for the software projects in the identified portfolio(s). This process is carried out either by gathering the source code of the projects, extracting the Gr^T values, and executing the tool implementing the AR^T rules locally, or by directly mining pre-computed AR^T and Gr^T values made available remotely (*e.g.,* if provided by contributors of the software portfolios, or a cloud service of the tool implementing the AR^T rules).

In **phase 4**, we execute the ATDx analysis in order to calculate the ATDD^T(SUA) and ATDx^T(SUA) for each project of the identified portfolios.

In **phase 5** we identify the relevant contributors of the selected software portfolio(s). Such contributors will then be contacted, in a following empirical evaluation phase, in order to gather insights into the obtained ATDx analysis results. Specifically, we are interested in contributors who are familiar with multiple software projects of the portfolio(s), in order to enable them to compare ATDx results across different projects. Hence, we select out of the all contributors of the software portfolio(s), the ones who contributed to at least two projects of the portfolio in the past 12 months.

Once we obtained the ATDx analysis results for each project of the portfolio(s), and established a curated set of contributors to be contacted, in **phase 6** we generate a personalized report for each contributor. Such report contains the ATDx results of each project of the contributor (represented as radar-charts) and further insights into the results (*e.g.,* architectural elements most affected by ATD).

In **phase 7**, the reports are shared with the contributors via a customized email, jointly with an invitation to participate to an online survey.

Finally, in **phase 8** we collect insights on the ATDx analysis results via an online survey. The survey is designed in order to require a short amount of time to be filled (this helps in terms of both response rate and participants fatigue). Moreover, various factors that influence response rates of developers are considered while designing the survey, such as *authority*, *brevity*, *social benefit*, and *timing* [104]. An overview of the questions composing the survey is reported in Table 3.1.

The survey is designed with a two-step approach. In the first step, a pilot version of

Table 3.1 – Survey questions

Question ID	Question Text	Response Type	Compulsory	Targeted RQ
Q1	How many years have you been developing software?	Integer	Yes	Demographics
Q2	How many open source software projects have you contributed to in your career?	1, 2-5, 6-10, >10	Yes	Demographics
Q3	On average, how familiar are you with the projects?	5-point Likert Scale	Yes	Demographics
Q4	By looking individually at each project: The radar-chart values reflect the project's current state of architectural debt	5-point Likert Scale	Yes	RQ1
Q5	By looking at all projects together: The radar charts reflect the differences in architectural debt present in the projects	5-point Likert Scale	Yes	RQ1
Q6	The architectural debt types displayed in the radar-chart are a good representation of architectural debt	5-point Likert Scale	Yes	RQ1
Q7	Do you miss any architectural debt type? If so, which one(s)?	Open-ended	No	RQ1
Q8	The results displayed in the radar charts inspire me to take action	5-point Likert Scale	Yes	RQ2
Q9	How would you use the radar-charts in your current practice?	Open-ended	No	RQ2

the survey is drafted and shared with 5 industrial practitioners within our personal network. In the second step, the questionnaire is reviewed and finalized by taking into account the collected feedback.

Questions Q1-Q3 assess the experience of the participants in terms of their experience (Q1-Q2) and familiarity with the open source projects included in the personalized ATDx report (Q3). To ensure the quality of the data gathered via the survey, survey responses of contributors not familiar with the projects included in their personalized report will be discarded. The subsequent 6 questions are designed to collect the data relevant to answer our RQs. Specifically, Q4-Q7 aim at assessing the core RQ of our study (RQ1), namely if the ATDx results are representative of the ATD present in the software projects. More in detail, with Q4 and Q5 we aim at evaluating if the inter- and intra-relations of the ATD present in software projects are preserved by the numerical relations calculated via ATDx [103]. With Q6 instead, we assess if the ATD dimensions ($ATDD^T$) identified during the building Step 2 of the ATDx approach (see Section 3.2.3.2) are a faithful representation of the overall ATD present in the software projects[6]. As a follow up to the previous question, Q7 investigates if any prominent ATD dimensions are missing in the used instance of ATDx. We opted to included two separate questions, Q6 and Q7, both focusing of ATD dimensions, in order to provide participants with a swift mean to provide input via a closed-ended question (Q6), while enabling them provide further details via the open-ended question (Q7). In order to evaluate if the ATDx results stimulate the active management of ATD (RQ2), we use the final two survey questions (Q8 and Q9). Specifically, with Q8, we directly assess the extent to which contributors are inspired to take action based on the ATDx results. With Q9 we gather further insights on the potential use of the ATDx in development practices. In addition to the questions reported above, the closed-ended questions targeting RQs (Q4, Q5, Q6, Q8) are supported by a complementary question ("*Comments?*"), allowing participants to add further detail into their closed-ended answer. In addition, the survey closes with a final complementary question ("*Do you have any final comments or suggestions?*"), designed to gather any additional input the participants may like to provide. To ensure that participants would be able to freely express themselves, an informative note is included in the survey invitation text, to assure them that all collected data would be anonymous.

The complete survey, comprising the aforementioned questions and supporting text clarifying terms and questions, is made available for review and replication in the publicly available supporting material of the paper.

[6] For the sake of clarity, in the survey, ATD dimensions are simply referred to as "ATD types".

In the following section, we report the details of our evaluation execution, which was conducted by rigorously adhering to the empirical evaluation design presented in this section.

3.4 Empirical Evaluation Execution

As shown in Figure 3.4, we executed the empirical evaluation by following the nine phases discussed in the previous section. In the following we give the details on the execution of each phase.

3.4.1 Phase 0: Selection of the SonarQube Tool

For this empirical evaluation we implement ATDx based on the SonarQube[7] static analysis tool. The rationale behind the adoption of SonarQube to implement the experimental ATDx instance is multifold: (i) SonarQube is widely used in industrial contexts [105], allowing us to have an ATDx instance potentially with high industrial relevance (which could be used by practitioners independently of our empirical evaluation), (ii) SonarQube is open-source, hence the source code of each of its AR_i^{SQ} rules can be inspected and associated to its granularity level Gr_i^{SQ} with relatively low effort, and (iii) the pre-computed SonarQube analysis results of several OSS projects are publicly available via the SonarCloud[8] platform, hence easing the AR_i^{SQ}(SUA) measurement retrieval process; those projects are actively maintained by several well-known organizations, such as the Apache Software Foundation[9], Microsoft[10], and the Wikimedia Foundation[11].

3.4.2 Phase 1: AR^{SQ} Identification and Classification

The goal of this phase is to establish the set of architectural rules AR^T from SonarQube . As input to this phase, we use a set R^{SQ} of Java-based SonarQube rules that were identified as design rules in a previous research [106]. Those rules represent

[7] https://www.sonarqube.org/
[8] https://sonarcloud.io/
[9] https://sonarcloud.io/organizations/apache/
[10] https://sonarcloud.io/organizations/microsoft/
[11] https://sonarcloud.io/organizations/wmftest/

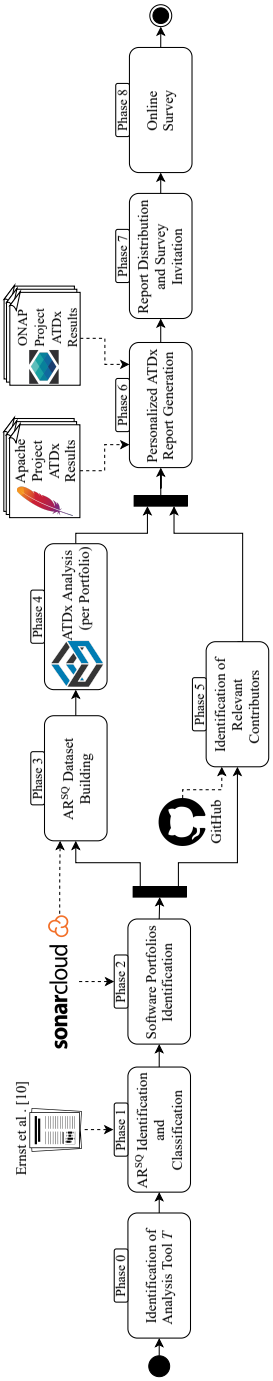


Figure 3.4 – Overview of the empirical evaluation of ATDx conducted in this study

a sound starting set of rules of potential architectural relevance, according to our definition of architectural rule presented in Section 3.2.1.

To select the architectural rules among the ones presented by Ernst *et al.* [106], we carry out a manual inspection of the definition of each single rule. Such inspection is based on the publicly-available official documentation of SonarQube [12]. The identification process is carried out by (i) analyzing the content of each rule description and (ii) evaluating it against the two criteria presented in Section 3.2.1. To mitigate potential threats to construct validity, two researchers independently carry out the identification. The identification process results in a 72.2% of agreement between the two researchers, with a substantial inter-rater agreement calculated via Choen's Kappa ($k = 0.62$). Then, a third researcher with several years of experience in software engineering takes over by (i) resolving possible conflicts and (ii) reviewing the the final set of architectural rules AR^{SQ}.

From the initial set of 72 SonarQube design rules presented by Ernst *et al.* [106], we identify 45 *architectural* rules. As detailed in Section 3.4.4, we further refine the set of identified rules during Phase 3 by removing the architectural rules which are not included in the SonarQube quality profiles[13] of the selected software portfolios. An overview of the final set of the architectural rules used in this study is reported in Table 3.2.

Once we established the set of architectural rules AR^{SQ}, we classify them in order to derive their associated granularity level GR^{SQ} and ATD dimensions ATDD^{SQ}, *i.e.,* we formulate the ATDx 3-tuples $\langle AR_i^{SQ}, Gr_i^{SQ}, \text{ATDD}_j^{SQ} \rangle$. This classification process is carried out collaboratively by three researchers, by discussing potential divergences till a consensus is reached among all researchers. Columns 1, 3, and 4 in Table 3.2 give an overview of the final mapping between the rules AR_i^{SQ}, their granularity Gr_i^{SQ}, and ATD dimensions ATDD_j^{SQ}.

Regarding the granularity levels Gr^{SQ}, we identify the four levels of granularity reported in column 3 of Table 3.2. The identified granularity levels are: *Java non-comment lines of code* (NCLOC), *Java method*, *Java class*, and *Java file*.

As for ATD dimensions ATDD^{SQ}, we elicited 6 core dimensions, namely *Inheritance*, *Exception*, *Java Virtual Machine Smell* (JVMS), *Threading*, *Interface*, and *Complexity* (see column 4 in Table 3.2). The *Inheritance* dimension (9 rules) clusters rules evalu-

[12] https://docs.sonarqube.org/latest/user-guide/rules/

[13] https://docs.sonarqube.org/latest/instance-administration/quality-profiles/

Table 3.2 – The architectural rules, granularity levels, and ATD dimensions used in the experiment

SonarQube ID	Short description	Granularity level (Gr^{SQ})	ATD Dimension ($ATDD^{SQ}$)
java:S107	Methods should not have too many parameters	Method	Interface
java:S112	Generic Exceptions should never be thrown	Java NCLOC	Exception
java:S1104	Class variable fields should not have public accessibility	Class	Interface
java:S1113	The Object.finalize() method should not be overriden	Class	Inheritance
java:S1118	Utility classes should not have public constructors	Class	Interface
java:S1130	Throws declarations should not be superfluous	Java NCLOC	Exception
java:S1133	Deprecated code should be removed eventually	Method	Interface, Complexity
java:S1161	@Override annotation should be used on any method overriding (since Java 5) or implementing (since Java 6) another one	Method	Inheritance
java:S1165	Exception classes should be immutable	Class	Exception
java:S1182	Classes that override "clone" should be "Cloneable" and call "super.clone()"	Class	Inheritance
java:S1185	Overriding methods should do more than simply call the same method in the super class	Method	Inheritance
java:S1199	Nested code blocks should not be used	Java NCLOC	Complexity
java:S1210	"equals(Object obj)" should be overridden along with the "compareTo(T obj)" method	Method	Inheritance, JVMS
java:S1217	Thread.run() and Runnable.run() should not be called directly	Java NCLOC	JVMS
java:S1610	Abstract classes without fields should be converted to Interfaces	Class	Interface
java:S2062	readResolve methods should be inheritable	Class	Inheritance
java:S2157	"Cloneables" should implement "clone"	Class	Inheritance, JVMS
java:S2166	Classes named like "Exception" should extend "Exception" or a subclass	Class	Exception
java:S2222	Locks should be released	Java NCLOC	Threading
java:S2236	Methods "wait(...)" "notify()" and "notifyAll()" should never be called on Thread instances	Java NCLOC	Threading
java:S2273	"wait(...)" "notify()" and "notifyAll()" methods should only be called when a lock is obviously held on an object	Java NCLOC	Threading
java:S2276	"wait(...)" should be used instead of "Thread.sleep(...)" when a lock is held	Java NCLOC	Threading
java:S2638	Method overrides should not change contracts	Method	Inheritance, JVMS
java:S2885	"Calendars" and "DateFormats" should not be static	Class	Threading
java:S2975	Clones should not be overridden	Class	Inheritance, JVMS

ating inheritance mechanisms between classes, such as overrides and inheritance of methods or fields. The *Exception* ATDD^{SQ} (6 rules) groups rules related to the Java throwable class "Exception" and its subclasses. *JVMS* (5 rules) contains rules which assess potential misuse of the Java Virtual Machine, *e.g.,* the incorrect usage of the specific Java class "Serializable". Rules associated with the *Threading* dimension (5 rules) deal with the potential issues arising from the implementation of multiple execution threads, which could potentially lead to concurrency problems. The *Interface* dimension (5 rules) encompassess rules assessing fallacies related to the usage of Java interfaces. Finally, the *Complexity* dimension (2 rules) encompasses rules derived from prominent complexity measures, *e.g.,* McCabe's cyclomatic complexity [107].

3.4.3 Phase 2: Software Portfolio Identification

Subsequent to Phase 1, we can proceed with the identification of software portfolios, *i.e.,* the experimental subjects of our empirical evaluation. In order to collect AR^{SQ} values, we opt to use the SonarCloud platform, which enables us to efficiently and effectively gather the data required for the ATDx analysis (see also Section 3.4.1). Hence, we want to identify portfolios that (i) are implemented in Java and (ii) make available pre-computed AR^{SQ} values via SonarCloud. In order to do so, we mine SonarCloud via its web-based API and (i) collect information about all public projects hosted on SonarCloud and (ii) identify the SonarCloud organizations[14] having the highest number of Java-based software projects. This leads us to identify two different software ecosystems, namely (i) Apache[15], covering general-purpose software components like the well-known Apache HTTP server, Apache Hadoop, and Apache Spark, and (ii) ONAP[16], focusing on orchestration, management, and automation of network and edge computing services. In this study we choose to target two different ecosystems as evaluation subjects[17] in order to mitigate possible external threats to validity. Indeed, focusing on Apache and ONAP allows us to study ATDx results for software portfolios developed for different contexts, and having different development processes, cultures, and technical backgrounds.

Among all the Java projects in each ecosystem, we filter out those without a corresponding GitHub repository. This filtering steps allows us to (i) have full traceability

[14] In SonarCloud, an organization is a *space where a team or a whole company can collaborate across many projects* (https://sonarcloud.io/documentation/organizations/overview).

[15] https://www.apache.org

[16] https://www.onap.org

[17] In the context of OSS, portfolios of OSS foundations like Apache are commonly referred to as "ecosystems" [108].

Table 3.3 – Summary statistics of the considered software projects

	Apache						
	Min.	Max.	Mean	Mdn	σ	CV	Total
Projects	-	-	-	-	-	-	126
Java NCLOC	90	383K	19.1K	2.9K	50.4K	2.6	2.3M
Java Files	5	4.4K	243.9	36	608.4	2.5	30.4K
Java Classes	3	4.6K	276.3	37	700.2	2.5	34.5K
Java Methods	21	34.9K	1.9K	241	5K	2.5	24.2K

	ONAP						
	Min.	Max.	Mean	Mdn	σ	CV	Total
Projects	-	-	-	-	-	-	111
Java NCLOC	753	239.9K	12.4K	5K	28.1K	2.3	1.3M
Java Files	10	3.6K	199.4	79	440.5	2.2	22.1K
Java Classes	9	3.2K	189.2	76	394.2	2	21K
Java Methods	49	22K	1.3K	518	2.8K	2.2	14.1K

	Total						
	Min.	Max.	Mean	Mdn	σ	CV	Total
Projects	-	-	-	-	-	-	237
Java NCLOC	90	383K	15.9K	3.7K	41.5K	2.6	3.6M
Java Files	5	4.4K	223	57	535.4	2.4	52.5K
Java Classes	3	4.6K	235.4	61	577.3	2.4	55.5K
Java Methods	21	34.9K	1.6K	352	4.1K	2.5	38.3K

Mdn: Median; σ: standard deviation; CV: coefficient of variation.

towards the source code of the system (useful for further inspections) and (ii) retrieve the names and email addresses of projects' contributors to be contacted for the survey (see Section 3.4.6). In order to avoid potential selection bias, we do not perform any other filtering step of the selected software projects, *e.g.*, by removing those with relatively low number of Java classes or few violations of the rules in AR^{SQ}.

The final set of software projects is composed of 126 Apache projects and 111 ONAP projects, for a total of 3.6 millions of non-commenting lines of Java code across 237 software projects. Table 3.3 and Figure 3.5 show the summary statistics of the selected

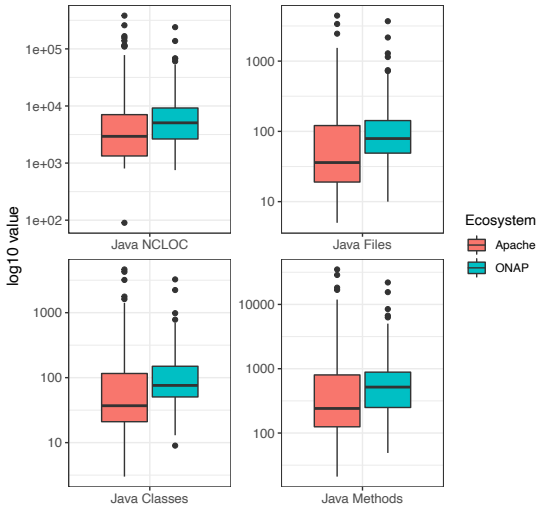


Figure 3.5 – Overview of identified ecosystems demographics

ecosystems. From Table 3.3, we can observe that the smallest software project is included in the Apache ecosystem, and is implemented by only 90 Java NCLOC. From Figure 3.5, we can see that this small project constitutes an outlier with respect to the other project of the ecosystem. In the ONAP ecosystem instead, the smallest software project is constituted by 753 NCLOC. The presence of small projects in the ecosystems is justified by the presence of "periferal" or "utility" software projects in the ecosystems, as further discussed in Section 3.6. The largest software project considered is also present in the Apache ecosystem, and includes 383K Java NCLOC. By considering the distributions reported in Figure 3.1, we observe that both ecosystems present some software projects possessing a high outlier size, that will considerably contribute to the total number of AR^{SQ} violations of the two ecosystems (as further detailed in the following section). Regarding the mean size of projects of the two ecosystems, we note that the ONAP ecosystem possesses overall projects of bigger size (cf. columns Mdn of Table 3.3). The median instead is higher in the Apache ecosystem, due to the presence in the ecosystem of some projects of exceptionally high size, as previously discussed. The variability in size is overall higher in the Apache ecosystems (cf. columns σ of Table 3.3), which is also reflected in the coefficient

Table 3.4 – Summary statistics of the mined AR^{SQ} values per software project

	Tot.	Min.	Max.	Mean	Mdn	σ	CV
Apache	17.4K	0	3.2K	139.5	12	457.2	3.2
ONAP	5.4K	0	616	48.9	11	99.9	2
Total	22.8K	0	3.2K	96.9	12	342	3.5

values (CV) of the two ecosystems.

3.4.4 Phase 3: AR^{SQ} Dataset Building

After the identification of the software portfolios, we proceed with building the dataset of AR^{SQ} values for each portfolio. As a preliminary activity, we check the SonarQube quality profiles used by Apache and ONAP in order to ensure that all rules in AR^{SQ} are included (see Phase 1). This activity led to the exclusion of 20 rules from the AR^{SQ} set; the final set of AR^{SQ} rules is presented in Table 3.2. This quality assurance step is needed only in the context of our empirical evaluation and it is necessary to ensure that all rules in AR^{SQ} rules contribute to the calculation of the ATDD^{SQ} and ATDx^{SQ} values.

After the consolidation of the AR^{SQ} setwe retrieve the AR^{SQ} values for each project included in the identified portfolios. This process is executed via automated queries to the SonarCloud API. We obtained a total of 22.8K AR^{SQ} rule violations across the 237 projects. For each rule violation, additional metadata is mined *e.g.,* the Java class where the violation occurs, the affected lines of code, and the textual description of the issue. Such information is then used for further analysis during the report generation phase (see Section 3.4.7), and is provided as complement to the ATDx report shared with the contributors.

An overview of the mined AR^{SQ} rule violations is reported in Table 3.4. As we can observe from the table, the total number of AR^{SQ} rule violations is much higher in the Apache ecosystem if compared to the ONAP one. We attribute this result to the presence, in the Apache ecosystem, to some large projects (cf. columns "Max." of Table 3.3), which are also characterized by a high number of AR^{SQ} violations (see column "Max." of Table 3.4). Both ecosystems include projects that do not present AR^{SQ} violations. The Apache and ONAP ecosystems have a median number of AR^{SQ} violations equal to 12 and 11, respectively. The standard deviation (σ) of

AR^{SQ} violations instead results much higher for the Apache ecosystem instead. As before, this result can be attributed to the presence of few projects of considerable size in the Apache ecosystem (see Figure 3.5). the same consideration can be made for the coefficient of variation (CV), as the projects belonging to the Apache ecosystem display a higher heterogeneity in sizes if compared to the ONAP projects.

3.4.5 Phase 4: ATDx Analysis

By following our empirical evaluation design, the ATDx analysis is run *independently for each portfolio, i.e.,* the analysis is based on the intra-portfolio comparison of AR^{SQ} values. As detailed in Section 3.3.2, this ensures that the clustering on which ATDx relies is executed by considering exclusively software projects sharing a similar context, hence reflecting the envisioned usage scenario of ATDx.

Figure 3.6 and Figure 3.7 give an overview of the ATDx analysis results. While the ATDD values vary across the projects of the two ecosystems, both of them exhibit low ATDx values (with a median of 0.28 for Apache, and 0.25 for ONAP). By considering the values of the various ATDD^{SQ} dimensions, we observe that none of them reaches the maximum of the scale (*i.e.,* 5 – see Section 3.2.2). This has to be attributed to the *potential empirically unreachable ATDD^{T} maximum values* property of ATDx, which is further discussed in Section 3.6. While it would be possible to convert the scale adopted in order to improve the presentation and intuitiveness of the results (*e.g.,* by converting local maxima to absolute ones), we refrain to do so, in order to support the transparency and understandability of the results.

Overall, we can observe that few dimensions contribute prominently to the cumulative value of the ATDx. Specifically, *Interface* is the dimension contributing the most to the overall ATD present in both portfolios, followed by the *Exception* dimension. The other ATDD^{SQ} dimensions contribute less in terms of ATD for both portfolios; nevertheless, some outliers are present, specially in the *Complexity* and *Exception* dimensions, meaning that some projects present an exceptional number of violations of rules belonging to such dimensions. Overall, the obtained results are in line with previous studies on other software metric indexes, *e.g.,* [109].

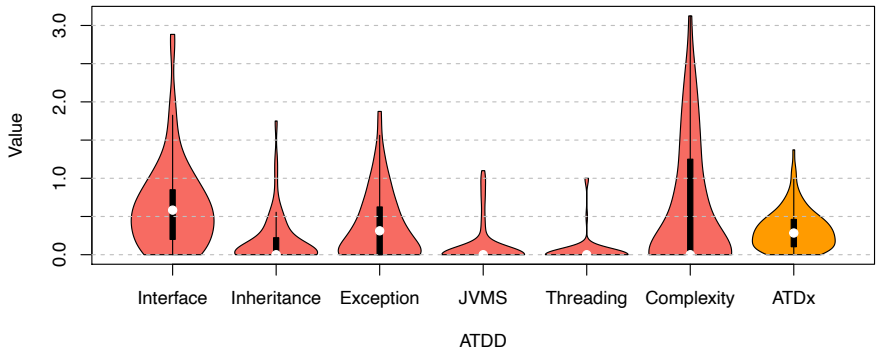


Figure 3.6 – ATDx analysis results for the Apache ecosystem

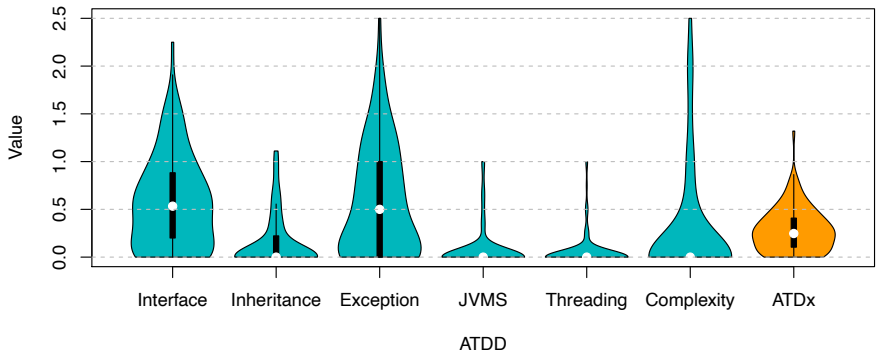


Figure 3.7 – ATDx analysis results for the ONAP ecosystem

3.4.6 Phase 5: Identification of Relevant Contributors

In parallel to the AR^{SQ} dataset building and ATDx analysis phases, we identify the relevant contributors for our study, *i.e.,* the contributors who will be invited to participate in our survey. As detailed in Section 3.3.2, we are interested in contributors who contributed to at least two software projects of a portfolio in the past 12 months. In order to identify such contributors, we first mine the GitHub repositories to identify contributors who pushed commits to the master branches of the software projects in the past 12 months. Subsequently, we identity all overlapping contributors, *i.e.,* contributors who resulted to be active in two or more projects included in a portfolio in the past 12 months. This process leads to the identification of 233 relevant contributors, 72 for the Apache ecosystem, and 161 for the ONAP ecosystem. No contributor is identified as a relevant contributor for both the Apache and ONAP ecosystems. For each identified relevant contributor, we store their contact information, along with the projects their contributed to, which will then be used to generate their personalized ATDx report in the subsequent phase of the empirical evaluation.

3.4.7 Phase 6: ATDx Report Generation

After the ATDx analysis, and the identification of the relevant contributors, we proceed with the generation of a personalized report *for each* relevant contributor. In total, 233 personalized reports have been generated. The generated reports follow the Markdown format and are hosted in a dedicated GitHub repository[18]. Using the Markdown format for the reports allows us to (i) show the ATDx results in a familiar environment for the projects' contributors and (ii) directly link the personalized report in the email inviting the contributors to participate to the survey.

An example of personalized report is shown in Figure 3.8. Each report is composed of three main parts, namely:

1. An introductory text providing the contributor with a concise explanation of the ATDx approach, the related background information (*e.g.,* a brief definition of the $ATDD^{SQ}$ dimensions), and a summary of the report content;
2. An overview of the ATDx analysis results for all projects of the contributor, provided in form of radar charts, to allow a swift comparison of ATDx analysis results across the projects;

[18] https://github.com/S2-group/ATDx_reports

Figure 3.8 – Example of ATDx report, comprised of ① concise description of the ATDx approach and related background information (*e.g.*, description of the ATDD^{SQ} dimensions), ② overview the project analysis results, and ③ per-project report, including the top-10 classes in terms of ATD violations.

3. A documentation of the ATDx analysis results for each project, including the top-10 classes containing the highest AR^{SQ} values, mapped to their ATDD^{SQ} dimensions.

Additionally, in order to provide the contributors with further context regarding the projects included in the report, each radar chart is complemented with additional information about the system under analysis, specifically: (i) a link to the original GitHub repository of the software project, (ii) a link to its SonarCloud dashboard, and (ii) a link to the complete raw data resulting from the ATDx analysis.

3.4.8 Phase 7: Report Distribution and Survey Invitation

After the generation of the reports, we share the results to the 233 relevant contributors identified in Phase 5. In addition to the distribution of the personalized reports, during this phase, we also invite contributors to participate to the online survey described in Table 3.1. Striving for a high response rate, we kept the invitation email as short and engaging as possible and customized its contents based on their receivers and the project they contributed to. Two different rounds of invitation, executed in two subsequent weeks, are used to stimulate the relevant contributors to participate in the survey.

3.4.9 Phase 8: Online Survey

In the last step of our empirical evaluation, we gather the data required to answer our research questions via the online survey. This survey is implemented by rigorously adhering to the structure presented in Section 3.3.2. We stop the data collection 4 weeks after the last round of invites are sent out. This allows us to finalize the results to be considered, while providing relevant contributors an adequate amount of time to participate to the survey.

Empirical Evaluation Setup. To evaluate ATDx, we implement an instance of the approach based on SonarQube, and 25 architectural rules derived from the literature. We run the ATDx analysis on two software ecosystems, namely Apache and ONAP (126, and 111 software projects, respectively). The analysis results are then shared, via personalized reports, to 233 contributors of the analysed project. Finally, we invited the 233 contributors to participate in an online survey designed to answer our research questions.

3.5 Results

In this section, we present the results of our empirical evaluation: Section 5.3.2.3 provides some demographic information regarding the participants of our survey; Section 6.3.4.1 reports on the results for RQ1, *i.e.,* the representativeness of the ATDx analysis results; and Section 6.3.4.2 documents the results for RQ2, *i.e.,* the extent to which the ATDx analysis results stimulate developers to take action with respect to the ATD detected in their projects.

3.5.1 Participants Demographics

In total, 47 out of 233 relevant contributors (20% response rate) participated in our survey. Participants have a median (average) of 12 (12.3) years of software development experience, with a minimum (maximum) of 2 (26). Most participants (97%) declared to have contributed to more than one OSS project, with the majority (38%) contributing to 6-10 OSS projects. All participants declared to be familiar with the analyzed software projects, with (i) 50% of them being very familiar with the projects (*i.e.,* "*occasional contributors*"), (ii) 40% of them being extremely familiar with the projects (*i.e.,* "*regular contributors*"), and 4% of them being moderately familiar (*i.e.,* "*have looked at its artifacts, read its code, and can contribute easily*"). Based on the gathered demographic data, we are reasonably confident that all participant have a good level of development experience and enough familiarity with the projects to properly understand the ATDx results shown in their personalized reports.

3.5.2 (RQ1) On ATDx Representativeness

In order to assess the representativeness of our approach, we examine the responses to questions Q4-Q7 of our survey (see Table 3.1). Figure 3.9 provides an overview of the responses given by the participants.

Question Q4 regards the extent to which ATDx analysis results reflect the actual ATD of a software project, by considering individually each project. The response distribution of this question reveals that most participants find the ATDx results representative (72%), with most participants agreeing with the statement formulated in Q4 (51%), or strongly agreeing with it (21%). Only a small portion of the participants does not find the ATDx results representative to various extents (8%), with only one participant strongly disagreeing with the statement. By considering the

Figure 3.9 – Response distribution of Likert scale survey questions used to answer our research questions (Q4, Q5, Q6, Q8)

rather sporadic open-ended comments provided by participants for this question (6 data points), we note a characteristic lack of awareness of the ATD present in their projects, *e.g.,* "*Not sure how much technical debt we have*" (4 data points), and a few acknowledgments of the results representativeness, *e.g.,* "*Results match my expectations*"(2 data points). Overall, by considering the answers provided by the participants, we conclude that ATDx is representative when individual projects are considered.

Question Q5 focuses on comparing ATDx analysis results across all projects within each personalized report. Based on the results gathered with this question, we note that most participants find the ATDx results representative when compared across projects, albeit to a lower extent than when considering the results of individual projects. In particular, while the majority of participants agrees with the representativeness of ATDx results (62%), answers are also characterized by a higher disagreement (11%), and the highest number of neutral answers among all Likert-scale survey questions (28%). The few comments provided by participants for this question (5 data points), point to difficulties in comparing the ATD across different software projects. We conjecture that the lower agreement with respect to Q4 could be attributed to inherent challenges in comparing the ATD present in different software projects, that would also motivate the high number of neutral responses measured for this question.

Question Q6 regards the representativeness of ATDD^{SQ} dimensions used in the empirical evaluation. By looking at Figure 3.9, we observe that overall Q6 yields the highest agreement rate (75%), and the lowest neutral (17%) and disagreement rates (8%). Rhe participants (7 data points) suggest in the open-ended comments: (i) adding more dimensions, (ii) adding specific dimensions (*e.g.,* "tests", "cloned code"), or (iii) adding more details about the dimensions already included.

The final question related to RQ1 (Q7 in Table 3.1) regards potentially-missing ATDD^{SQ} dimensions of the specific ATDx instance used in the empirical evaluation. This question is optional and only three participants answered it. Nevertheless, the provided answers are informative and propose the following additional dimensions: "duplicated classes", "testing", and "cloned code".

Main findings (RQ1, ATDx Representativeness). The survey results confirm the representativeness of the ATDx analysis results. The representativeness of the dimensions used in the ATDx instance implemented for this empirical evaluation present the highest agreement rate (75%), followed by the representativeness of the analysis results within individual projects (72%). The comparison of analysis results across different projects is characterized by the (relatively) highest portion of disagreeing and neutral responses (respectively 11% and 28%), potentially due to inherent difficulties in comparing architectural debt present in different software projects.

3.5.3 (RQ2) On ATDx Actionability

With RQ2 we aim to assess the degree to which the ATDx analysis results stimulate developers to take action towards addressing their ATD. In the survey, this RQ is covered by two separate questions: a Likert scale question (Q8) and an open-ended question (Q9).

As shown in Figure 3.9, participants generally agree with the statement that the results displayed in the radar charts inspire them to take action (60%). The remaining participants tend to either disagree with the statement (21%) or take a neutral stance (19%). By considering the additional comments provided by the participants to support their answer (6 data points), we observe the need for a finer-grained level of information in the ATDx report to address the identified ATD, as the provided documentation may not be sufficient to trigger concrete action on the analysis results. Examples of requested additional information include: *"Give more information on problems"* and *"Add more technical debt aspects"*.

Based on this finding, we conclude that the current information documented in the ATDx reports (namely ATDD^{SQ} values, top classes with AR^T violations, and JSON files containing the raw SonarQube analysis results) is perceived as actionable. Nevertheless, participants also suggested interesting points for improvement, *e.g.,* by providing (i) the ability to zoom in and out of ATD hotspots at different levels of abstraction, (ii) ATD visualizations, (iii) hints about ATD resolution strategies.

Question Q9 is about the scenarios in which the ATDx analysis can be used in practice. Even though only 10 participants answered this question[19], participants mention

[19]The low response rate for question Q9 might be due to the question being formulated as optional, and asked at the end of the survey.

some interesting usage scenarios about visualization (*e.g.,* "*as a UI in SonarQube*"), refactoring (*e.g.,* "*find code to fix*" code review), and communication (*e.g.,* "*talk about problems in issue tracker*"). Also, participants highlight the lack of a user interface to visualize the analysis results in an interactive manner. Driven by the results collected for RQ2, we envision to improve the reporting of ATDx analysis results, in order to improve its actionability, and directly support a set of selected usage scenarios, *e.g.,* by enabling the composition of the analysis with continuous integration pipelines, issue trackers, and enabling a finer-grained scrutiny of the result via a dedicated dashboard.

Main findings (RQ2, ATDx Actionability). The ATDx results tend to be actionable, with usage scenarios including refactoring, code review, communication, and ATD evolution analysis. Points for improvement include the need to provide more informative reports and the lack of an interactive dashboard.

3.6 Discussion

With our empirical evaluation, we gathered different insights regarding the *in vivo* application of ATDx . Overall, the empirical results demonstrated the representativeness of the approach and, even if to a lower extent, its actionability. Regarding the empirical evaluation, it is important to note that the results are bound to an experimental implementation of ATDx, and hence have to be considered only as a proxy of the general ATDx approach presented in Section 3.2. Nevertheless, implementing an ATDx instance is an inevitable step required to evaluate the approach. This leads to a potential threat to validity of our findings, as further discussed in Section 6.3.5.

Implementing a concrete instance of ATDx allowed us also to gain further hands-on knowledge of the characteristics of the approach. Specifically, when considering the approach benefits, we took advantage of the (*by design*) *tool independence* of ATDx, allowing us to use a readily available rule set [106] and the pre-computed measurements of SonarCloud.

The *language independence* property of ATDx instead allowed us to focus on the software portfolios deemed best fitted for the empirical evaluation, rather than having to follow potential constraints dictated by other analysis approaches.

As described in [54], the semantic metric aggregation on which ATDx is based, allows

to provide *multi-level granularity results*. This characteristic of the approach was used in the empirical evaluation by including in the ATDx report architectural ATDD^T dimension values at project-level, AR^T rule violations at class-level, and localization of single AR^T rule violations at line-of-code-level.

Actionability of ATDx resulted to be lower with respect to its representativeness (see Section 6.3.4.2). We conjecture that this result did not depend considerably on the adopted levels of granularity, but rather on how the analysis results were documented in the ATDx report. As future work, we look forward to refine the ATDx report capabilities, which were only marginally considered for this investigation, by providing enhanced visualizations of analysis results (*e.g.,* via dashboarding), and information on how to resolve the identified ATD issues.

The empirical evaluation conducted in this study provided us also further insights on the *data-driven* nature of ATDx, *i.e.,* its reliance on inter-project measurement comparison, rather than predefined metric thresholds. This led to the establishment of two severity classification frameworks tailored *ad-hoc* for the two portfolios considered, implementing different empirically-derived severity thresholds.

Some of the envisioned benefits of ATDx described in [54] could instead not be assessed with our empirical evaluation design. Prominently, the ATDx instance was based on a single tool, namely SonarQube. This did not allow us to study the *tool composability* property of ATDx, *i.e.,* the aggregation of analysis results gathered via heterogeneous tools. As future evaluation of the ATDx methodology, we plan to assess the effects of tool composition on the ATDx analysis results.

In addition to tool composability, we did not conduct any *domain-specific customization* of ATDx, other than filtering out the AR^{SQ} rules that were not included in the SonarQube quality profiles of Apache and ONAP. While the Apache and ONAP ecosystems could be deemed as oriented towards specific domains (namely web servers, and networking/edge-computing respectively), upon further inspection we noted a high heterogeneity across projects of the same ecosystem. The heterogeneity of projects belonging to the same ecosystems has to be attributed to "periferal" and "utility" software project, that support the general domain of the organizations, but focus on narrow use cases or implementation concerns. As an example taken from the ONAP ecosystem, the ONAP SO project[20] implements a core functionality in the ONAP domain, namely the orchestration of ONAP components. The ONAP

[20] https://github.com/onap/so

SDC[21] instead is a "utility" project, which does not focus directly on networking or edge-computing, as it implements a visual modeling and design tool. To carry out a fine-grained domain customization of ATDx, a curated portfolio including *exclusively* software projects belonging to a specific domain, *e.g.,* safety-critical systems or mobile applications, should be considered.

Regarding other characteristics of ATDx presented in [54], in this work we directly addressed the *emphasis on outlier values*, characteristic of the previous version of ATDx. To overcome this limitation, we substituted the *outlier* function used to calculate the constituent values of ATDD with the *severity* function (see Formula 3.4), which is based on the CkMeans clustering algorithm. This adjustment allowed us to calculate ATDD values at a refined level of granularity, by determining the severity of each AR^T violation of software project, rather than focusing on a boolean characterization of its outlier violations.

For this research, we also carried out an *ATDx implementation validation*, required to assess the representativeness of an implemented instance of ATDx. Rather than utilizing focus-groups, as envisioned in the publication this research builds upon [54], we leveraged personalized reports and follow-up surveys (see Section 3.3.2): this allowed us to contact in an efficient way a considerable number of developers who contributed to the analyzed software projects.

The ATDx approach is dependent, by definition, on a portfolio of software projects. Hence, while numerous ATD analysis approaches require exclusively the SUA [1], our approach needs instead a portfolio of software projects to calculate the severity of AR^T violations. This implies that the ATDx results of a SUA are only "relative" to the other projects included in the portfolio, and are not representing an absolute result. As a consequence, the ATDx analysis results are not directly comparable across different portfolios. For example, by considering the distribution of ATDD violations of Figures 3.6-3.7, it is important to remember that the results are intrinsically dependent on the portfolios considered. As a consequence, the distributions presented in Figure 3.6 and Figure 3.7 are not directly comparable. The ATDx dependency on a portfolio can be considered as both a benefit and a drawback of the approach. On one hand, ATDx resolves potential problematics related to statically defined metric thresholds and debt values *via* severity clustering, allowing to fine-tune the ATDx analysis based on the specific portfolio and context considered. On the other hand, the approach can be utilized exclusively if a portfolio of software projects is available.

[21] https://github.com/onap/sdc

Another characteristic inherent to the ATDx design is the *potential empirically unreachable ATDx maximum values*. While reaching a maximum value in a certain dimension ATDD^T is by definition theoretically possible, it is empirically extremely improbable. By design, a software project reaches the maximum in one dimension if and only if it possesses maximum severity values in *all* ARs mapped to a ATDD^T dimension. If the software project possesses a maximum value of ATDD^T, this would indicate that the project is characterized by exceptionally severe and recurrent issues in that dimension. In our empirical evaluation, such project was not present for any dimension. As reported in Section 3.4.5, a possible heuristic to improve this characteristic of ATDx would be to rescale the values of a dimension i to the $[0, max_i]$ range, where max_i is the maximum value in the dimension i across all rules in AR_T mapped to i. Nevertheless, we refrained from such solution in order to support the transparency and interpretability of the results.

Building an ATDx instance entails a human-in-the-loop (as implied in the second building step of ATDx, see Section 3.2.3.1). In fact, the classification of 3-tuples relies on manual classification, and hence is inherently characterized to a certain extent by subjectivity. Despite our best efforts to document a systematic classification process, mitigation mechanisms should be adopted in order to reduce potential sources of bias during the execution of this step (*e.g.,* by involving different individuals in this step, and systematically tracking inter-rater agreement levels).

Finally, a last characteristic of ATDx is its reliance on a predefined set of AR^T. As detailed in [54], ATDD values are computed by considering distinct sets of ARs. It is necessary that the number of rules across the different sets is balanced as, if the distinct sets exhibit notable differences in cardinality, the weight of under-represented sets could lead to their unfair representation. In the ATDx instance utilized in our empirical evaluation, this characteristic was meticulously considered and mitigated by carefully selecting AR^T from existing academic literature [106], and by considering the AR^T recurrence, relevance, and the cardinality of the mapped ATDDs.

Overall, our empirical evaluation shows how ATDx can be a valuable approach to gain awareness of the ATD present in a software-intensive system. The approach can be tailored to the specific context one considers, by utilizing measurements gathered via the tools available, and relate the severeness of architectural rule violations with respect to other similar projects included in a software portfolio. The ATDx report results provide an intuitive yet meaningful overview of ATD, which can be enhanced via further visualization techniques to provide actionable guidance of ATD hotspots and their resolution.

3.7 Threats to Validity

Despite our best efforts, the presented results could suffer from potential threats to validity. Following the classification of Wohlin *et al.* [73], we consider four different threat types.

3.7.1 Conclusion validity

Conclusion validity regards if the experimental measurements are measuring the theoretical constructs they are intended to measure. As the results of our empirical evaluation are gathered via a survey, a possible thread to conclusion validity is the face validity of the survey [110], *i.e.,* the extent to which the survey conveys the concept it purports to measure. In order to mitigate potential threats to face validity, supporting text explaining the goal of the survey, concepts related to ATD, and the ATDx approach purpose and functioning, were integrated in all material shared with the survey participants, namely the survey invitation message, the ATDx report, and the survey itself. Additionally, to ensure that the survey questions were sufficiently clear to participants, and no important aspect was missing in the questions, each closed-ended question was accompanied by an open-ended question ("*Comments?*"), were participants could add clarifications to their answers, doubts, and remarks.

To avoid potential threats related to the extent to which the survey answers are fitted to answer our RQs, we designed the survey questions by deriving them directly from the RQs and the goal of our empirical investigation. This led to the formulation of different survey questions, covering the various aspects the RQ purported to assess. To ensure full traceability of the mapping between survey questions and RQs, the complete process leading to the formulation of each survey question is documented in Section 3.3.2, while the explicit mapping between survey question and RQ is also schematically reported in Table 3.1.

Potential threats related to low statistical power are mitigated by documenting separately the distributions of answers to each single question of the survey, allowing for independent scrutiny and interpretation of the gathered results. Additionally, the 20% response rate results to be aligned with other survey-based software engineering investigations [111]. Hence, we are confident that this threat may have only marginally influenced our results, if at all.

3.7.2 Internal validity

Internal validity regards if the observed results are actually due to the treatments. Regarding the experimental subject utilized for the ATDx analysis, *i.e.*, the Apache and ONAP ecosystems, we note that the projects considered may also contain non-Java source code, even if tagged as "Java" software projects on SonarCloud. To mitigate potential threat to internal validity, we consider for the ATDx analysis exclusively SonarQube rules pertaining to Java. To avoid instead potential bias when selecting the AR^{SQ} rules, three researchers were involved in Step 1 of the *ATDx* building process, their level of agreement was measured statistically, and disagreements were jointly discussed with the help of a third researcher. The same mitigation strategy was also applied for the definition of the Java-based 3-tuples in step 2.

Regarding the survey adopted, an internal threat to validity regards the potential influence that the invitation, ATDx report, and survey text may have had on survey answers. In order to mitigate this threat, all text was kept as neutral and formal as possible. Additionally, survey participants were informed that the survey was completely anonymous, so to allow them to freely express themselves.

Another threat to internal validity regards the extent to which the survey participants understand the concept of ATD, and its difference with TD. This background knowledge is required to ensure that participants correctly interpret the survey questions, and do not base their answers on approximate or erroneous assumptions. In order to mitigate this threat, at the beginning of both the personalized reports and the survey we present to participants (i) background information on ATD and (ii) how we defined it in the context of this study. In addition, survey questions were formulated in order to be as straightforward and intuitive as possible. Each question was supported by a free form text field in order to allow (i) the participants to express their potential doubts and (ii) us to gather additional insights into their answers.

A threat to validity which could have affected our results regards *maturation* [73]. Specifically, the positioning of optional open-ended questions towards the end of the survey may have influenced their response rate. Nevertheless, we prioritized clarity and flow of the survey over this potential threat, and mitigated it by ensuring that answering all survey questions would require as little time as possible.

Potential selection biases were mitigated by defining *a priori* a rigorous selection process to identify the portfolios and survey participants used in the empirical evaluation (see Section 3.3.2). Additionally, to ensure the soundness of the selected participants, a set of demographic questions, including the familiarity with the shared software

projects, was included in the survey.

Finally, a last threat to internal validity regards the evaluation method adopted, namely blind survey.

3.7.3 Construct validity

Construct validity regards if our empirical evaluation is appropriate to answer the RQs. A prominent threat to construct validity, presented in Section 3.6, regards the evaluation of a specific instance of ATDx in order to evaluate the approach. As this step is required, we could not completely avert this threat, which has to be considered while interpreting the results. To mitigate its influence, we based our ATDx implementation on a widely popular static analyser, a starting set of design-related rules presented in the academic literature [106], and two prominent OSS ecosystems, one of which was already adopted in various other TD studies [112, 113, 114, 115].

The adoption of a blind survey to evaluate our approach could constitute another threat to construct validity, as we did not posses any knowledge of the participants identity, other than the mined OSS data, and the demographic data gathered via the preliminary survey questions (see Table 3.1, Questions Q1-Q3). Adopting a blind survey however allowed us to mine data from a considerable number of real-life software projects (237), *i.e.,* maximizing participants thanks to anonymization.

Another potential threat to construct validity is constituted by the adoption of OSS software ecosystems as portfolios for the ATDx analysis. To mitigate potential threats related to the selection of the portfolios, we ensured that they included a considerable number of software projects (237 in total), belonged to established OSS organizations (Apache and ONAP), and utilized SonarQube in their continuous integration pipeline. Additionally, for our survey we selected exclusively participants who contributed to at least two software projects of the portfolios, hence allowing them to compare ATDx analysis results across different projects.

In order to mitigate potential threats to mono-operation and mono-method bias [73] in our survey design, we formulated it as a mix of open-ended and closed-ended questions, with different questions mapped to each RQ.

3.7.4 External validity

External validity regards whether and to what extent our observations can be generalized. A potential threat to external validity concerns the representativeness of the portfolios selected for our empirical evaluation. As reported in the previous section, we mitigated potential threats to external validity by ensuring that only relevant portfolios, and their contributors, were considered. In addition, the tool on which our experimental instance of ATDx is based, namely SonarQube , is one of the most frequently used static analysis tools for Java-based software projects [105], making us reasonably confident about the relevance of its rules in real-world projects. Despite our best efforts to mitigate external validity threats, such could potentially influence our obtained results, especially if proprietary portfolios or other source code analysis tools are considered. Future research will naturally further strengthen the external validity of the results reported in this research, *e.g.,* by experimenting with ATDx in industrial settings, and by considering additional source code analysis tools.

3.8 Related Work

In this section we discuss the academic and industrial work related to this study. Specifically, we consider as related work approaches aimed at detecting ATD, approaches aimed at providing indexes of ATD and TD, and additional work that share conceptual similarities with ATDx.

Regarding approaches aimed at identifying ATD, numerous software analysis approaches have been proposed during the years. Among the most prominent and current ones, the approach of Arcelli Fontana *et al.* [116], Martini *et al.* [117], and Roveda *et al.* [118] focus on the identification of ATD by analyzing dependency architectural smells, which could lead to the emergence of an additional ATDD dimension, namely *"Dependency"*. Similarly, Kazman *et al.* [119], Xiao *et al.* [93], and Cai *et al.* [120] analyzed ATD by inspecting antipatterns of semantically related architectural components, *e.g.,* by the analysis of bug-prone components. Building on the notions presented in such previous studies, in a follow-up research, Cai *et al.* [121] introduce DV8, a tool designed to measure software modularity, detect architecture anti-patterns, and quantify the maintenance cost of each anti-pattern instance. Among the most prominent differences, ATDx deviates methodologically from the approaches presented in studies reported above by utilizing inter-project severity clustering and semantic aggregation of violations into different ATD dimensions. Another related study of Nord *et al.* [122], differentiating from ATDx for the same

reasons, presented an ATD metric based on rework associated to changing dependencies of architectural components and values of features delivered. Among the studies considered so far, the most closely related one is the work of Roveda *et al.* [118] as it presents another ATD index. Differently from ATDx, this index focuses on architectural smells, notably related to dependency violations.

Le *et al.* instead reported an empirical investigation of architectural decay via the analysis of 8 architectural smells of different nature [123]. Interestingly, in such study, smell violation severity is evaluated by adopting interquartile analysis [124], similar to the first iteration of the ATDx [54]. As a further difference with ATDx, the analysis proposed by Le *et al.* utilizes *intra* architectural rule level analysis and values are not normalized *per* system-size.

More ATD identification approaches are reported in a secondary study of Verdecchia *et al.* [1], albeit none of the included primary studies present an ATD index, with exception of the work of Roveda *et al.* previously discussed [118]. In another related survey study, Arcelli Fontana *et al.* [125] present a preliminary discussion on technical debt indexes provided by tools. In contrast to the other studies considered so far, the work of Arcelli Fontana *et al.* focuses on proprietary tools. Among these, the tools that share most commonalities with ATDx are CAST[22], inFusion[23], Sonargraph[24], and Structure101 [25]. While such tools focus to various extents on ATD and provide indexing capabilities, they are conceptually different with respect to ATDx. In fact, the calculation of the indexes implemented in such tools is generally based on the multiplication of violations' recurrence times the effort required to fix the violations, rather than relying on a data-driven and inter-project comparison-based severity calculation. For example, the index provided by CAST is calculated by multiplying the number of rule violations times the criticality of the rules violated times the effort required to fix the rule violations. With ATDx, we distantiate from *a priori* defined rule severity and remediation effort, as recent literature pointed towards their potential inaccuracy [126]. An additional proprietary tool is the Software Analysis Toolkit (SAT) developed by the Software Improvement Group (SIG) [96]. Such tool, similar to ATDx, is intended to carry out software portfolio quality monitoring. Nevertheless, the implementation details, internal workings, and metrics used do not appear to be disclosed to the public. To the best of our knowledge, SAT differentiates from ATDx in multiple ways, *e.g.,* it is not a tool-agnostic approach (as it implements its

[22] http://structure101.com/products/workspace/
[23] http://www.intooitus.com/, which evolved in the tool AI Reviewer http://www.aireviewer.com
[24] https://www.hello2morrow.com/products/sonargraph
[25] https://structure101.com/

own quality metrics and rules), and does not consider clustering for dynamic issue severity classification.

Regarding the identification of metrics thresholds, similarly to the first version of ATDx, Alves *et al.* [127] adopt an interquartile strategy to identify the severity of metric values. As additional differences, such study does not focus on ATD and, while adopting a system-size normalization strategy, it considers only one level of granularity (NCLOC). Finally, in a recent work, Ulan et al. [128] proposed a software metric aggregation approach based on their distribution. Our approach is different by (i) adopting a clustering algorithm to determine violation severity, (ii) considering sizes according to distinct granularities, and (iii) clustering results into different semantic dimensions.

3.9 Conclusions and Future Work

Over the years, numerous approaches have been proposed to detect ATD instances present in software intensive-systems. Such methods rely on the analysis of symptoms through which ATD is manifested, and consider *ad-hoc* definitions of technical debt and software architecture, in order to best fit the conceived analysis processes. When ATD indexes are provided by approaches and proprietary tools, they are most commonly based on formulae considering *a priori* defined values, such as severity, remediation cost, and metric thresholds, that are potentially prone to human estimation and approximation inaccuracies, and disregard the context of the analyzed software. Furthermore, to the best of our knowledge, such indexes do not aim at providing an encompassing view of the (potentially highly heterogeneous) ATD present in a software-intensive system, but rather focus on a specific facet of ATD.

To fill this gap, in this research we presented ATDx, an approach leveraging the analysis of a software portfolio, pre-computed architectural rule violations, and granularity levels, to compute severity levels of ATD violations via a clustering-based algorithm. Results of ATDx are aggregated into a purely data-driven index, which is composed of different "ATD dimensions", providing information on the facets of the ATD measured.

In order to evaluate the representativeness and actionability of ATDx, we implemented an instance of the approach based on SonarQube, and run the analysis on two software ecosystems, Apache and ONAP. We then shared the results with targeted contributors, and invited them to participate in a survey designed to collect their

feedback on ATDx.

The gathered answers showed that ATDx analysis results are representative, especially when considered for each project individually, and that the used ATD dimensions are an indicative representation of ATD. Results also showed the actionability of the approach, although to a lower extent when compared to the ATDx representativeness.

The collected results are promising, but we deem this investigation as a preliminary step towards the consolidation of ATDx. As future research activities, we envision to mitigate potential threats to validity associated to our results by conducting further empirical evaluation by considering also *e.g.*, proprietary portfolios, different programming languages, source-code analysis tools, and software domains. Additionally, based on our findings, we envision to enhance the reporting capabilities of results, in order to strengthen its actionability, and directly support a set of selected usage scenarios, *e.g.*, by enabling the composition of the analysis with continuous integration pipelines, issue trackers, and providing a finer-grained scrutiny of the results via a dedicated dashboard.

In conclusion, with ATDx we do not aim at providing a "silver bullet" to identify the ATD present in a software-intensive system: the multifaceted nature of ATD comprises a plethora of different ATD items, symptoms, causes, and consequences, which hinders a holistic general-purpose approach. Rather, with ATDx we strive for the establishment of a sound, comprehensive, and intuitive architectural view of code-related ATD, which helps facilitate conversations, understanding, and awareness of the current state of ATD in software-intensive systems.

4 Architectural Technical Debt: A Grounded Theory

When the past is always with you, it may as well be present; and if it is present, it will be future as well.

William Gibson, *"Neuromancer"*

This chapter is based on:
R. Verdecchia, P. B. Kruchten, and P. Lago, *Architectural Technical Debt: A Grounded Theory* [56], European Conference on Software Architecture (ECSA), 2020.
R. Verdecchia, P. B. Kruchten, and P. Lago, I. Malavolta, *Building and evaluating a theory of architectural technical debt in software-intensive systems* [57], Journal of Software and Systems (JSS), 2021.

This chapter documents the investigation carried out in order to answer the third research question of this thesis (RQ3). Specifically, this chapter reports a grounded theory study, executed by leveraging interviews with experienced software practitioners in order to establish a grounded theory on architectural technical debt. The results provide a comprehensive theory on architectural technical debt, considering the multifaceted dimensions of related phenomena. Emerging results include, among other categories, grounded findings relative to architectural technical debt items, their causes, consequences, symptoms, prioritization problems, and management strategies. Conforming to grounded theory principles, the categories are not presented as isolated instances, but are instead presented as a unified theory thanks to the analysis of the relations among the categories which constitute the theory.

Contents

4.1 Introduction

Technical Debt (TD) is a concept that has been with us for a long time, at least since 1992 when Cunningham crafted the phrase [129], but it only got some real attention from researchers in the last 10 years [130]. What is technical debt? "In software-intensive systems, technical debt consists of design or implementation constructs that are expedient in the short term, but set up a technical context that can make a future change more costly or impossible. Technical debt is a contingent liability

whose impact is limited to internal system qualities, primarily maintainability and evolvability" [131].

Technical debt can take many different forms in software development, and can be found in many different places [132]. While much of the literature and tooling available today address code-level technical debt, our focus is on Architectural Technical Debt (ATD). This is the technical debt incurred at the *architectural level* of software design, that is, in the decisions related to the choice of structure (*e.g.,* layering, decomposition in subsystems, interfaces), the choice of technologies (*e.g.,* frameworks, packages, libraries, deployment approach), or even languages, development process, and platform. As software systems grow in size and their lifespan extends to many years, many of these original design choices become constraints, and limit future evolution or even prevents it. To evolve the system, developers do find workarounds and often complicated solutions, which introduce quality issues and delays. Large and long-lived systems are suffering from architectural debt, while the small and short-lived ones die before ATD becomes a real problem. For example, a research prototype may work well for its intended goal, but if used as brittle architectural foundation for a commercial product, it can lead to the failure of a company after years and years of strenuously accumulating workaround on workaround.

However, despite its importance and widespread presence, as of today *our knowledge of ATD is still incomplete.* Indeed, how to accurately identify, monitor, and manage ATD is to date still an open question. The **goal** of this paper is to fill this gap by providing novel insights of the crucial factors which characterize ATD in industry. In order to achieve this goal, in this study we applied a mixed-method empirical strategy based on the *grounded theory* method and *focus groups.* This strategy allows us to (i) systematically organize and report in a cohesive theory the knowledge acquired by experienced practitioners on the topic and (ii) evaluate and refine the emerging theory according to the new data collected via the focus groups.

The **main contribution** of this study is the development and evaluation of an ATD theory, which provides an empirically-extracted conceptualization of the architectural technical debt phenomenon. For example, we identified architectural issues, their symptoms, and managements strategies, which not only shed light on the state of the practice on ATD, but also provide means for researchers and practitioners to further understand and monitor ATD phenomena. While the focus of our theory is on the architecture of software-intensive systems, the emerging results can be utilized to create specializations of the theory by considering a different abstraction level, e.g., code-level technical debt.

In a previous study [133] we reported a preliminary version of our theory for ATD. In this paper we extended our previous work in multiple ways:

- we added 2 new categories, 24 new concepts, and introduced a new inter-level abstraction of concepts, referred to as *Type*. These new results emerged thanks to a further in-depth theoretical coding process, by analyzing the relations between substantive codes, and how these were represented in terms of concepts, categories, and relations between them;
- we conducted an evaluation of the theory by adopting the focus group method, which led to the assessment of the theory according to a set of predefined criteria, and the introduction of 12 additional concepts in the theory;
- we added an in-depth discussion of the related work by analysing how the emerging ATD theory complements the findings and visions of existing studies on architectural technical debt.

The **target audience** of this study includes practitioners and researchers. Our theory provides a solid foundation which benefits (i) *practitioners* aiming at a better management and mitigation of the ATD they experience, and (ii) *researchers* looking for precise and evidence-based definitions of ATD-related concepts, which may in turn help exploring new research directions towards a better characterization of ATD and its effective management.

The paper is structured as follows. The next section focuses on providing background on the grounded theory method, followed by specifics of our study design and execution. Section 6.3.4 reports the results of our investigation, with each of the Subsections 4.3.1- 4.3.10 dedicated to the description of a specific category of our theory. Related work is reported in Section 4.4, while threats to validity to our study are reported in Section 6.3.5. Section 6.3.7 concludes the paper.

4.2 Research Method

The research strategy followed in this study consists of two separate parts, carried out subsequently. Specifically, in the first part of the investigation, in order to formulate a theory on ATD, we adopted a grounded theory method (see steps Ⓐ - Ⓗ of Figure 4.1). Afterwards, once the theory on ATD was established, we applied focus groups in order to evaluate and refine our theory (see step Ⓘ of Figure 4.1). The remainder of this section gives an overview of the complete research process utilized

in this investigation. We structure this section as follows: Section 4.2.1 summarizes the grounded theory method, Section 4.2.2 documents the grounded theory design and execution, including the details about data collection and data analysis, and Section 4.2.3 details the focus group method adopted to evaluate and complement the emerging theory.

4.2.1 Grounded Theory

To build a theory on Architectural Technical Debt we adopted Grounded Theory (GT), a qualitative research method enabling us to establish a theory by grounding our findings in the experience of software practitioners. GT is used to systematically explain an observed phenomenon by studying how people conceptualize and deal with it in practice. As summarized by Schreiber *et al.* [134], the goal of grounded theory is to answer the question "What is going on here?". To do so, *incidents* (*i.e.*, bits of gathered data related to the studied phenomenon) are analyzed to identify emerging *concepts*. As the research progresses, the growing number of concepts are aggregated semantically into different *categories*, which constitute the basic building blocks of the emerging theory. Categories are further developed by gathering additional data and comparing the new incoming incidents against the old ones, which were already categorized. This inductive process leads to the identification of abstract categories, which are theoretically shaped by letting their definition fit all of the underlying data. The iterative data collection and analysis process stops once the identified categories become *saturated*, *i.e.*, when new data is no longer triggering their revision or reinterpretation. In addition to the identification of the categories constituting a theory, GT requires to analyze incidents to identify the conceptual relationships existing between the different categories. In fact, a theory established by using GT is not mere taxonomy or "set of themes", but rather a cohesive set of constructs and relationships describing the studied phenomenon.

An overview of the GT research process followed in this study is depicted in Figure 4.1. It starts with a **bootstrap question** which drives the whole study and reads as follows.

Which architectural design decision do you regret the most today?

Then, the method is based on the following concepts:

Ⓐ **Theoretical Sampling.** New data is collected iteratively by purposely identifying current gaps and/or unsaturated categories of the theory. Theoretical

Figure 4.1 – Overview of the grounded theory research method (Ⓐ - Ⓗ), evaluated via focus groups (Ⓘ)

sampling guides the selection of new data sources (*e.g.,* participants), and the data to be collected (*e.g.,* by generating iteratively interview questions).

Ⓑ **Coding.** Incoming data is processed by subdividing it into incidents (*e.g.,* single lines of text, or paragraphs), and subsequently labeling the incidents with analytical codes summarily expressing their semantic meaning. Codes are then compared and further analyzed by considering their properties in order to infer theoretical concepts and categories of the emerging theory.

Ⓒ **Theoretical Sensitivity.** The data gathering and analysis processes are guided by theoretical sensitivity. This concept refers to the creative ability of the researcher, and guides the theoretical sampling, conceptualization of incidents, and identification of relations between concepts.

Ⓓ **Memoing.** Throughout the entirety of a grounded theory study, *memos* (*e.g.,* textual notes, sketches, diagrams) are taken. Memos are used to keep track of emerging concepts/categories, the relations between them, and potential gaps in the theory. Memos are constantly compared to the emerging theory and new incidents, in order to ensure that the categories best fit the underlying data. This latter process is referred to as *memo sorting* (or *theoretical sorting*).

Ⓔ **Constant comparison.** Throughout the entirety of the study, all artifacts (incidents, codes, concepts, categories, and memos) are constantly compared and updated. This process is executed to ensure that the emerging theory is cohesive, and is coherent with the underlying data in which it is grounded.

Ⓕ **Theoretical Saturation.** The data collection and analysis process terminates once the categories become *saturated*, *i.e.,* when adding new data does no longer result in an update of the established theory.

Ⓖ **Literature Review.** In GT studies, a comprehensive review of the literature is commonly postponed till after the establishment of the theory. Limiting the researcher's exposure to the literature is crucial to ensure that the emerging theory is grounded in the collected data, and is influenced as little as possible by preconceptions and already established concepts.

To date, three prevailing versions of GT can be found in the literature, namely Classic (or Glaserian) GT, Straussian GT, and Constructivist GT. They mainly differ in three aspects, namely philosophical point of view (objectivism, pragmatism, and social constructionism, respectively), coding procedures (open-selective-theoretical coding,

open-axial-selective coding, and initial-focused-theoretical coding), and role of the literature (while all stances acknowledge the general guideline presented in Ⓖ, they differ in other related details, as further discussed in [135]).

4.2.2 Grounded Theory Design and Execution

For this study, we adopted the classic "Glaserian" method [136], and we conformed to it throughout the whole study, from data collection, to data analysis and synthesis, with the exception of our adoption of a different "coding family" than the ones suggested by Glaser [137], as explained in Section 4.2.2.2. In divergence to the other GT stances, during the analysis process of the method described by Glaser [137], a "core category" is established. The core category captures the most variation in the data [138] while addressing the main concern of the study participants (see item Ⓗ of Figure 4.1 to position the discovery of the core category within the research method of this study.) The "Glaserian" GT method provided us with the ability to gain a fresh and independent viewpoint on ATD, by letting concepts emerge from the experience of our participants, rather than from preconceived views of researchers. The principal investigator was not too immersed in the technical debt world prior to this study, and avoided doing an extensive review of the literature on ATD prior to the data analysis, thus minimizing possible confirmation biases, and improving his "theoretical sensitivity" [139]. As prescribed by Glaser [140], we delayed this review of the literature after our theory emerged, in order to avoid the influence of existing concepts on the theory. Prior to starting our investigation, we studied the fallacies and guidelines for grounded theory in software engineering research presented by Stol *et al.* [141], in order to avoid common pitfalls, and ensuring the soundness of our methodology throughout the study. The investigation, including data collection, data analysis, and reporting, lasted approximately 6 months.

4.2.2.1 Grounded Theory Data Collection

To collect data, we conducted semi-structured interviews with industrial practitioners. Participants were recruited first by convenience and then by following theoretical sampling: we contacted initial participants within our personal network, and then selected further based on gaps in the emerging theory, or to investigate unsaturated concepts. This lead us to interview 18 experienced practitioners, with a mean industrial experience of 17.5 years, from 14 distinct companies in different industrial domains. We identified via theoretical sampling senior technical leaders as best

Table 4.1 – Grounded Theory Participant Demographics

Id	Role	Ex	Domain	CS	CId
P1	Senior Vice-President of SE	21	Banking	S	C1
P2	Software Staff Engineer	17	Telecom	M	C2
P3	Senior Director of SE	20	Enterprise Software	XXL	C3
P4	Chief Technology Officer	14	Financial Services	M	C4
P5	Senior Software Engineer	22	Health	L	C5
P6	Senior Software Engineer	8	Software Tooling	M	C6
P7	Senior Software Engineer	18	Software Tooling	M	C6
P8	Senior Software Engineer	23	Software Tooling	M	C6
P9	Vice-President of Product	15	Data Analysis	M	C7
P10	Senior Software Engineer	12	Software Tooling	M	C6
P11	Senior Director of Technology	26	Data Technologies	M	C8
P12	R&D Director	27	Enterprise Software	L	C9
P13	Senior Software Engineer	14	Software Tooling	M	C10
P14	Senior R&D Manager	16	Enterprise Software	L	C9
P15	Chief Software Architect	11	Cloud Services	M	C11
P16	Chief Technology Officer	12	Consultancy	S	C12
P17	Co-Founder	33	Consultancy	XS	C13
P18	Founder	22	Mobile Applications	XS	C14

Id: Participant identifier; Role: current role of participant; Ex: industrial experience (in years); CS: company size (XS<20; S<100; M<500; L<5K; XL<10K; XXL>10K); CId: Company identifier.

fitted participants for data collection, given their hands-on experience on a vast range of ongoing (and concluded) long-lived software projects. Table 4.1 presents an overview of the participant demographics. Interviews lasted approximately 1 hour and were conducted face-to-face at the practitioner's workplace, or for a few via Skype video-calls when it was not possible to meet in person due to geographic distance.

As the emerging theory should guide the sampling process, we solved the "bootstrap problem" [142] of GT by starting our first interview with the bootstrapping question described in Section 4.2.1. Then, the other interview questions emerged iteratively by following theoretical sampling, in order to let participants express their main concerns on ATD in their own words. Specifically, the data collection was conducted in the form of "guided conversations" [143], *i.e.,* in the form of unstructured questions, formulated to investigate unsaturated concepts emerging in our theory, or gain

further details on concepts described by the participants during the interview. As advised by Rose [144], during the interviews we refrained to influence the scope or depth of the responses, as doing so could have influenced the data collected, and lead to the inclusion into the theory of preconceptions of the researchers on the topic of ATD. We deemed the use of unstructured interviews to collect data as best fitted for our GT investigation. In fact, unstructured interviews allowed respondents to use their own way of defining the world, by assuming that no fixed sequence of questions is suitable for all respondents, enabling participants to raise considerations the interviewer did not consider [145].

In addition to the unstructured questions, we also utilized a predefined set of demographic questions to collect data on the professional background of participants, such as current role, and years of industrial experience (see Table 4.1).

Interviews were audio-recorded and transcribed manually by following the denaturalism approach, that is, grammar is corrected, interview noise (*e.g.,* stutters) is removed and nonstandard accents (*i.e.,* non-majority) are standardized, while ensuring a full and faithful transcription [146].[1]

The data collection terminated once we reached theoretical saturation, that is, when components of our theory are well supported and new data is no longer triggering theory revisions or reinterpretations [139]. Figure 4.2, which displays the slow increase of cumulative codes w.r.t. the number of participants, shows that we have achieved this theoretical saturation around participant number 16.

4.2.2.2 Grounded Theory Data Analysis

We followed Glaser's grounded theory data analysis and synthesis processes to create our theory: open coding, selective coding, and theoretical coding [136, 139]. Specifically, we examined the whole body of text transcripts, subdivided them into separate incidents [136], and labeled them with codes to let the theory concepts emerge. When possible, codes are generated by directly quoting the incident (*e.g.,* see [S-Q1]). Otherwise, "synthetic" codes summarizing the semantic meaning and emerging concept of the incidents were created by the authors. Subsequently, concepts were clustered into fundamental descriptive categories, which guided the future data collection. Finally, we established the conceptual relations between the different emerging categories,

[1] An initial *ad-hoc* automated solution resulted to be too literal, *e.g.,* by including repeated portions of sentences, inconclusive sentences, etc., leading to lenghty transcripts, which would have impacted negatively the subsequent data analysis.

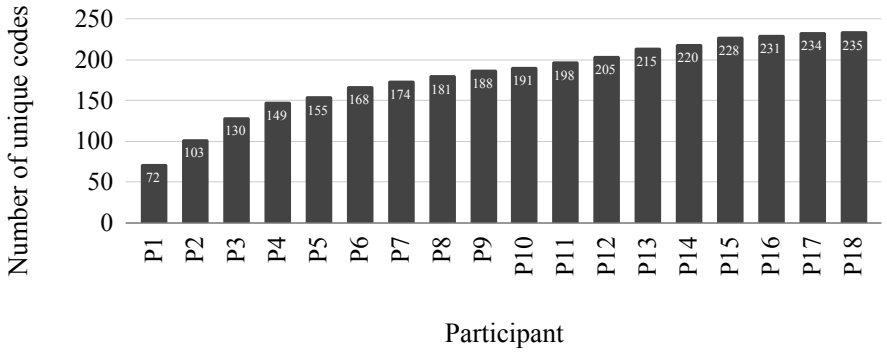


Figure 4.2 – Cumulative unique number of codes per participant, showing theoretical saturation

leading to the formulation of our theory. We express the relationships between codes as hypotheses via a UML model to precisely describe the relations of different nature emerging between the categories of our theory (see Figure 4.6).

Differently from Glaser, who used "concept" and "category" as synonyms, we associate to such terms two distinct levels of theoretical granularity, as also done in numerous studies utilizing GT, *e.g.,* [142] and [147]. When required, we use an additional abstraction level, referred to as *Type*, which aggregates distinct concepts of similar nature in a mid-ranged level of abstraction. The identification of types was conducted during the theoretical coding phase, when concepts were taken into account. Specifically, when similar characteristics shared among concepts of the theory emerged in the memos, a new *type* was instantiated, according to the identifying characteristic shared among the underlying concepts. In summary, our theory entails four different levels of abstraction, ranked from lower to higher abstraction level: *code*, *concept*, *type*, and *category*. An example of such abstraction hierarchy, regarding the concept of *symptom* is reported in Figure 4.3.

During the entirety of the coding procedures, we made use of *memoing* [136]. We created textual memos to elaborate concepts (i) related to single incidents (*e.g., "This incident exemplifies the impossibility to implement new functionality due to ATD"*) and (ii) orthogonal to multiple incidents (*e.g.,* relations between concepts or categories, such as *"Developer's intuition can lead both to ATD identification and prioritization"*).

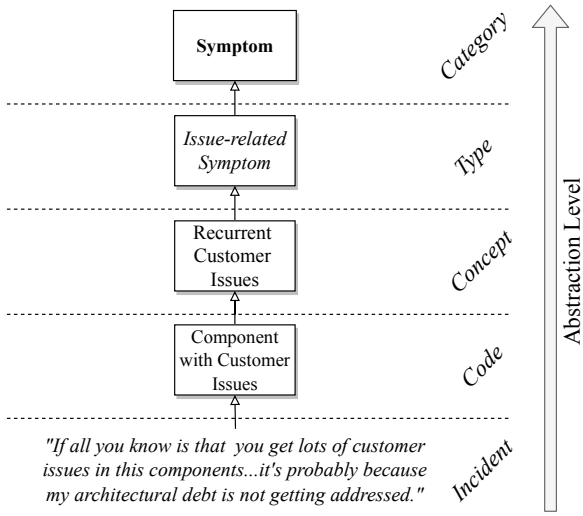


Figure 4.3 – Example of Abstraction Levels of the ATD Theory

In addition, we adopted word clouds to gain a concise overview of the codes emerging from each interview. An example of these type of memos is shown in Figure 4.4.

As described in Section 4.2.2.1, we analysed our data immediately and continuously, using simultaneous data collection and analysis, guided by theoretical sampling. Additionally, during data analysis, we constantly compared our data, memos, codes, and categories, in order to identify and keep track of common notions, topics, and patterns, as they emerged. Similarly, we continuously sorted our memos to evolve the emerging concepts and categories to best fit our codes, leading to the formulation of a substantive, cohesive theory. We performed continuous comparison until additional data being collected did not add new knowledge about the categories, *i.e.,* until we reached the state of saturation (see Section 4.2.2.1).

It should be noted that numerous concepts included in our theory possess a multi-faceted nature. For instance, by considering the concept of "technical debt" itself, we can observe how it can be both a *cause*, leading to the introduction of additional debt, and a *consequence*, *e.g.,* of pre-existing debt which is accumulating. By following GT principles, we coded multifaceted concepts according to the facet which was

Figure 4.4 – Example of memo used to gain a summary overview of the codes emerging from a single interview

deemed most important by participants. This process was adopted in order to ensure the emergence of concepts from the data gathered, rather than from preconceived knowledge of the authors. Coding incidents via this strategy allowed the emergence of issues of importance to the participants to be exposed from their own point of view, systematically uncovering patterns of which participants might even not be aware of [148].

Four researchers were involved in both the data collection and analysis phases, where the first author carried out the coding, memoing, and analysis processes, while the others collaboratively analysed and reviewed the obtained results through several iterations.

4.2.3 Theory Evaluation via Focus Groups: Design and Execution

In order to evaluate and refine our theory after its emergence, we applied the focus group method [149] (item Ⓘ in Figure 4.1). This step of our research consisted of presenting the theory to groups of industrial practitioners, and gathering feedback based on their discussion to evaluate and complement the theory. Specifically, the

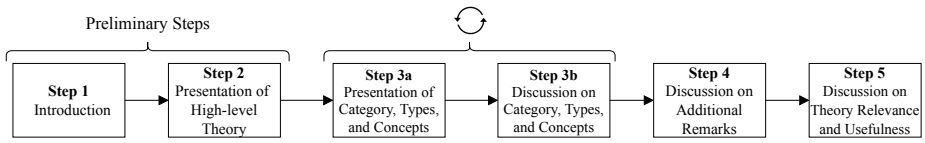


Figure 4.5 – Main steps of the focus group sessions

evaluation of the theory was conducted by following the criteria characteristic of the Glaserian GT method [139], as we employed such stance of GT to construct our ATD theory, namely:

C1: the categories of the theory *fit* the underlying data;
C2: the theory is able to *work* (*i.e.,* explain and reason about ATD related phenomena);
C3: the theory has *relevance* to the domain (*i.e.,* development and management practices of large and long-lived systems);
C4: the theory is *modifiable* as new data appears.

We adopted the focus group method to evaluate the theory as it enabled us to effectively and efficiently gain feedback on the theory by allowing participants to compare their experiences, jointly discuss opinions on it, and release potential inhibitions with respect to the discussed phenomenon.

Table 4.2 – Focus Group Participant Demographics

FG	Id	Role	Ex	Domain	CS	CId
FG1	P19	IT Architect	15	Banking	XL	C15
FG1	P20	Principal Architect	26	Consulting	L	C16
FG1	P21	Director Enterprise Architecture	27	Airline Industry	XXL	C17
FG1	P22	Vice-President	34	Consulting	XXL	C18
FG2	P23	ICT Business Manager	25	Finance	L	C19
FG2	P24	Product Owner, Integration Architect	13	Airline Industry	L	C20
FG2	P25	Enterprise Architect	31	Banking	XL	C15
FG2	P26	Enterprise Architect	36	Finance	M	C21
FG2	P27	Director	25	Consulting	XS	C22

FG: focus group identifier; Id: Participant identifier; Role: current role of participant; Ex: industrial experience (in years); CS: company size (XS<20; S<100; M<500; L<5K; XL<10K; XXL>10K); CId: Company identifier.

As shown in Figure 4.5, each focus group session was organized in five distinct steps. During Step 1, the purpose of the focus group was presented, and some background knowledge on architectural technical debt was given, to set a general common ground on the topic guiding the subsequent discussion. Additionally, during this first step, a round of introduction among the participants and moderators was conducted, to give participants confidence to speak up, provide context for the experiences described by them in the subsequent steps, and foster group dynamics. In the second step, a high-level overview of the theory, presenting the categories of the theory and their relations (see Figure 4.6), was introduced. Then, a deep dive into each category of the theory was conducted. This process consisted in comprehensively presenting each category, its related types, and concepts (Figure 4.5, Step 3a), followed by a discussion among the participants about the topic presented (Figure 4.5, Step 3b). During the discussion of each category (Step 3b), the conversation was guided by the moderator to assess if (i) the theory reflected the experience of the practitioners, (ii) any prominent information was missing in the theory, and (iii) the theory contained new or unexpected categories, types, or concepts. Step 3a and Step 3b were repeated for each category of the theory. After all categories were covered, *i.e.,* the theory was discussed in its entirety, participants were given the possibility to express further remarks on the complete theory (Figure 4.5, Step 4). While Steps 3-4 focused primarily on assessing the GT evaluation criteria C2 and C4 (*i.e.,* the theory *works* in practice, and is *modifiable* according to new data), the last step of the focus group (Figure 4.5, Step 5) was designed to assess the GT evaluation criterion C3, *i.e.,* if the theory is *relevant* to action in the area of ATD, by focusing on the emerged core categories and concepts [150]. Specifically, this last step consisted in a discussion among participants about the relevance of the theory they perceived, as well as the potential usage scenarios of the theory they envisioned. In order to prepare participants, and ensure that they were well informed of the focus group goal and content, a document describing the theory and the structure of the focus group was provided to them two weeks prior their session.

Table 4.2 gives an overview of the focus group participant demographics. The participants of each focus group session were selected by ensuring a balance of commonalities and differences in their expertise, to ensure a range of variegated opinions, while sharing the common background knowledge required to discuss and compare experiences and opinions. Like for the grounded theory participants, we selected for the focus groups practitioners expert in the area of software architecture, as a deep knowledge of the ATD phenomenon is a crucial characteristic, especially in order to get insightful feedback on the ATD theory. In total, 9 participants were identified and assigned to one of the two separate focus group sessions used for this study. We opted

to conduct two separate sessions, as this allowed us to avoid flat group dynamics, while ensuring that each participant had sufficient time to express their opinion [151]. Focus group sessions lasted approximately 1.5 hours, and were conducted virtually.

In the following section, we document the theory resulting from the execution of the GT method, refined with the feedback from the focus groups. Further considerations on the focus group evaluation are reported in Section 4.5.

The emerging theory is the product of both grounded theory and focus groups methods. For the sake of traceability, concepts included in the theory due to discussions emerging in a focus group session are denoted with a characterizing icon (👥).

4.3 A Theory of Architectural Technical Debt

Figure 4.6 gives an overview of our grounded theory on Architectural technical Debt (ATD). In this section we describe the categories emerging from our data, which constitute the foundation of our grounded theory on architectural technical debt.

The **system** category represents the system being developed. In this research we follow the definition of "software-intensive system" as defined in the ISO/IEC Standard 42010, *i.e.,* "any system where software contributes essential influences to the design, construction, deployment, and evolution of the system as a whole" [152]. A system possesses a certain amount of architectural technical debt.

The **ATD** category embodies the entirety of the technical debt incurred at the architectural level in a software-intensive system. Regarding the definition of technical debt, in this research we follow the 16162 definition, *i.e.,* "a collection of design or implementation constructs that are expedient in the short term, but set up a technical context that can make future changes more costly or impossible" [131].

In addition to reporting the categories of our theory theory, in this section we also discuss the relations emerged between the different categories. In line with the grounded theory approach, this enables us to both present comprehensively the emerging theory, and offer explanations underlying ATD related phenomena [139] [153].

At the core of our theory lies ***ATD item***, *i.e.,* the category that embodies the instances of ATD residing in a software-intensive system (for an in-depth description of this category, see Section 4.3.1). The identification of the *ATD item* as the core category of our theory can be observed from the numerous relations between this category and

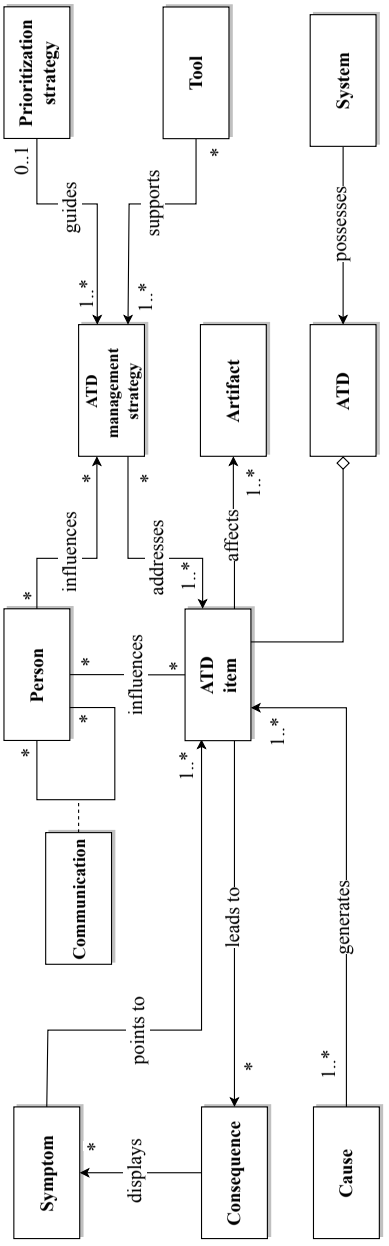


Figure 4.6 – Core categories of the ATD theory and their relations [133]

the other ones reported in Figure 4.6.

At the root of each ATD item lies one or more ***cause***. Each cause can *generate* one or more items (see Section 4.3.2. From our data time pressure and business drive are the main causes leading to the generation of ATD items:

"*The plan is one thing, but it's not working now, we have to adapt quickly. Whether or not we meet the coding rules, I proceed. I don't care. Something is broken, nobody cares how nicely something fits the architecture, I care if it's gonna break our product. That is not a computer science issue, it's a business one.*" - P8, Senior Software Engineer [R-Q1]

As causes can generate one or more ATD items, so ATD items can lead to one or more ***consequences***, *e.g.,* reduced development velocity, higher maintenance cost, impossibility to implement new functionality (see Section 4.3.3). Additionally, in contrast to the relation between *Cause* and *ATD item*, ATD items can also be "dormant", *i.e.,* the items are present in the system, but do not lead to any immediate consequence:

"*There was a developer who wrote a component that nobody knows how it works, and so we are all afraid of touching it. It works well for now, but if something stops working, or we have to touch that, for example to implement some new functionality, we could have a problem.*" P12, R&D Director [R-Q2]

Consequences can *display* one or even multiple ***symptoms***, *e.g.,* recurrent customer, performance, and/or development issues. In this case, a consequence could also not display any symptom, either because the related ATD item is "dormant", or because the observed symptoms are not sufficiently distinct to establish the relation:

"*To be honest? I have a bit of a vibe. As a product manager, I'm pretty like face-to-face and hands on, and I kind of just gauge the winds on the face of developers*" P9, Vice-President of Product [R-Q3]

Symptoms *point to* one or more ATD items, *i.e.,* observing symptoms displayed by a *consequence* can lead to the identification of one or more *ATD items*. Often, a multitude of symptoms point to a single, widespread, ATD item:

"*You do things like: "How are your bugs?", "How is your performance?". All of those things tell you something. They are indicators. Like code coverage, it tells you something, but does it really tell you anything? But it's just one big underlying problem!*" P3, Senior Director of Software Engineering [R-Q4]

Nevertheless, as reported in quote [R-Q3], consequences of ATD items can also not

display any clear symptom, making the discovery of related ATD items harder.

Each ATD item can *affect* one or more ***artifacts***, *e.g.,* software components, test suites, software development tools, and/or documentation:

"*We reached the point where it [architecture] became quite brittle, and it was also quite difficult to change the test suite, because the architecture was so complex...so many connectors...and the variance of those connectors!*" P7, Senior Software Engineer [R-Q5]

Similarly, an ATD item can *reside* in one or more artifacts, *i.e.,* it can be present simultaneously in various artifacts of different nature, or even occur in the relation established between two or more artifacts.

ATD items can be *addressed* via one or more ***ATD management strategies***, *e.g.,* via systematic time allocation, large-scale rewrites, and/or carry out opportunistic patching (see Section 4.3.5). Additionally, it is also possible to address multiple ATD items with a single management strategy (typically via rewrites):

"*Usually, I just do a gut evaluation: if there is a large disconnect between what the system does and what it is supposed to achieve, usually it is a big indicator that there are many problems, and we need a rewrite.*" P1, Senior Vice-President of Engineering [R-Q6]

ATD management strategies can be guided by a **prioritization strategy**, *i.e.,* a strategy with which ATD management tasks are prioritized along with other development tasks, such as bug fixes, and implementation of new functionality [154] (see Section 4.3.8). Often, prioritization processes are not carried out systematically, and can consider one or multiple management strategies depending on the addressed ATD item(s):

"*Given three weeks of development time, which architectural debt should we pay down? I would say, we're not doing it systematically, but we're probably not coming out with two very different answers. If something is really painful, we would know.*" P9, Vice-President of Product [R-Q7]

ATD management strategies can also be supported by ***tools***, *e.g.,* static analyzers and linters, such as Clang Tidy[2] and SonarQube[3]. Nevertheless, only in unique instances practitioners used tools to detect architectural debt issues, such as component de-

[2]https://clang.llvm.org/extra/clang-tidy
[3]https://www.sonarqube.org

pendency anti-patterns via NDepend[4]. In most of the cases, ATD management strategies are not supported by any tools, possibly due to their perceived immaturity or usefulness:

"The really expensive type of debt [ATD], I have not seen a tool which is able to detect that..." P10, Staff Software Engineer [R-Q8]

An emerging category which is directly related to the *ATD item* category is **person**. The relation between *person* and ATD items is of a multifaceted nature, as people's personal drive, skill set, and awareness (among other concepts, see Section 4.3.9) can highly influence ATD items, from their establishment to their prioritization, and resolution.

ATD can lead to the **communication** of concepts related to it among people working on a software-intensive system where ATD is present. This constitutes another emerging category of our theory, it is reported in Section 4.3.10.

Numerous relations of secondary nature between categories were also identified in our theory. To maintain the documentation of our theory compact, such complementary relations are discussed through the support of cross-references in Sections 4.3.1-4.3.10, further relating concepts and categories via exemplifying incidents (*e.g.,* [S-Q3] not only discusses an ATD symptom, but also hints to the inability of solving complex ATD issues via the described ATD management strategy, namely *opportunistic patching*).

4.3.1 ATD Items

An overview of the ATD items residing in software-intensive systems which emerged in our theory is depicted in Figure 4.7. The relation between elements of Figure 4.7 has to be interpreted as a "is a type of" relation (same applies for Figures 4.8-4.11, and Figure 4.14). The identified items belong to one of three mutually exclusive types, namely *framework ATD*, *process ATD*, and *implementation ATD*[5]. Framework ATD items are specific to the adoption and adaptation of software frameworks in software projects. Process ATD items, instead, regard the high-level processes of architecting and managing software-systems, with particular emphasis on their evolution. Finally, implementation ATD items focus on lower-level implementation details which, due to their widespread impact on the maintenance and evolution of a software-system,

[4] https://www.ndepend.com/

[5] In the next figures, **categories** are shown in bold, *types* in italics, and concepts as plain text.

become of architectural relevance. The remainder of this section is dedicated to the description of each concept belonging to the ATD Item category.

4.3.1.1 Framework ATD items

Unfitted Framework. One of the most prominent ATD items related to software frameworks regards the adoption of a framework which is misaligned with either the currently implemented architecture or its requirements. This ATD item is often caused by a lack of comprehensive trade-off analysis of alternative frameworks. As P14 described:

"*We had a discussion on how to build the new front-end in React. At the time there were reasons that supported our decision, but later on we saw that we didn't evaluate all the options.*" P14, Senior R&D Manage [ATDI-Q1]

This type of item is often incurred inadvertently. Additionally, its consequences mostly manifests themselves only over a prolonged period of time, increasing the effort required to maintain and evolve an architecture containing an inadequate framework, potentially embedding the framework deeper into the architecture. As pointed out by P1:

"*The technology decision sounded great in theory, but in practice it was a real pain. At the time it felt like a good idea, but in the long run, the cognitive overhead to deal with that solution led to a lot of pain, bad code, bugs, and additional effort.*" P1, Senior Vice-President of SE [ATDI-Q2]

Re-inventing the Wheel. This ATD item refers to *ad-hoc* components developed in-house, which are chosen over already available components with similar functionalities (*e.g.,* components available as open source software):

"*We basically built our own thing . . . why would we build our own persistence library? That doesn't make sense! It's just silly!*" P11, Senior Director of Technology [ATDI-Q3]

As noticeable in the previous quote, re-inventing the wheel ATD items are particularly evident when generic functionalities, widely available as open-source software, *e.g.,* the mentioned persistence library, are re-implemented in-house.

In addition to the resources required to implement already available components,

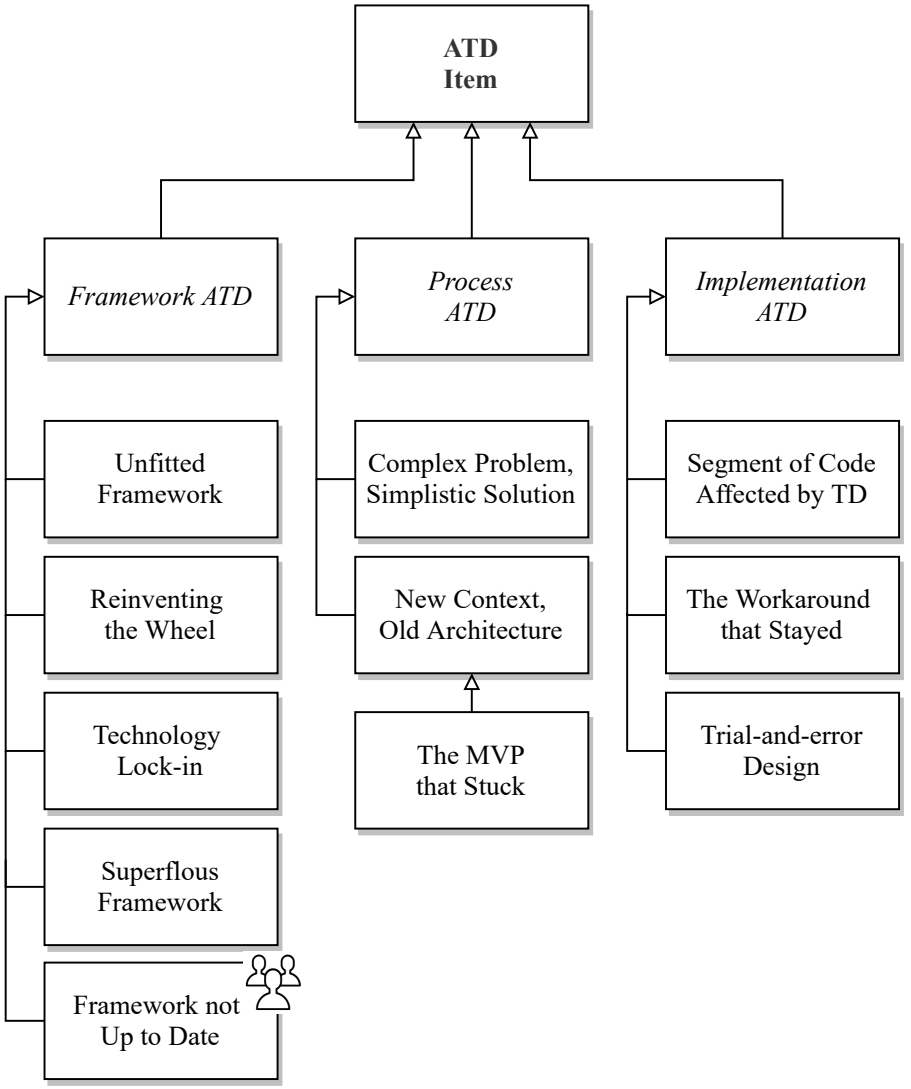


Figure 4.7 – Overview of emerging ATD items

drawbacks include lower implementation quality, additional maintenance cost, and lack of documentation:

"We built our own thing . . . and now it's hard to maintain. And now that we have got to build on top of it, people are getting tired" P8, Senior Software Engineer [ATDI-Q4]

Ad-hoc components are often chosen due to the perceived velocity of developing a new component instead of getting accustomed to, and adapting, an existing one. Additionally, as further discussed in Section 4.3.9, personal drive of developers can influence this decision:

"I thought to be smarter, but I was not . . . in the long run, off-the-shelf solutions make people faster in ramping up, even if you [just] have to adapt them." P3, Senior Director of SE [ATDI-Q5]

"People had NIH-Syndrome, not-invented-here [laughs]." P10, Senior Software Engineer [ATDI-Q6]

Framework Lock-in. Related to the previous debt item, ATD can arise due to software frameworks which, due to their deep embedding into the architecture of a software-intensive system, become very costly or even impossible to replace. This debt item is often referenced as harmful if co-occurring with "dormant" ATD items [R-Q2], or if the lock-in is of technological nature and unreliable (*e.g.*, a third party has complete ownership of a component and releases a breaking change). As described by P1:

"Sometimes you make something overly-specific, lock in completely into a specific library or technology. It's about how able your system is to change without crystallizing in design choices dictated by the need of adaptation." P1, Senior Vice President of SE [ATDI-Q7]

An example of framework lock-in was provided by P11, regarding the data layer of a software-intensive system they worked on. Specifically, during the evolution of their architecture, SQL became deeply embedded throughout their system. As the system grew in size, due to scalability concerns, the passage to a NoSQL database was required. Nevertheless, as the architecture was completely locked-in on SQL, the system had in the end to be deprecated.

Superfluous framework. Due to its uncertainty, the process of building up technical credit (see Section 4.3.9) can lead to the achievement of the opposite of its goal, namely the introduction of new ATD items. In relation to frameworks, efforts spent in

gaining technical credit can lead to the adoption of superfluous frameworks. Superfluous frameworks are characterized by often complex and hard to embed technology solutions, which implement numerous functionalities that will never be used. P12 described one of such occurrences, referring to the adoption of Apache Tomcat as web server environment:

"In hindsight we didn't need it. There was a lot of functionality we could have used, but it was not useful, and in the end and we didn't use it. Wait till you need it, then worry about it. Thinking about it now it was just overkill . . . there is no need to go for the moon when you need to go into the sky." P12, R&D Director [ATDI-Q8]

The adoption of superfluous frameworks can be due to the inherent optimism bias characterizing developers (cfr. Section 4.3.9), and the often misplaced assumption that more complex and expressive solutions are generally better than simple ones. P1 recalled:

"We thought the solution we chose was more expressive, so it should be better. We assumed we could be able to deal with a more complex thing at the time. We thought that more complexity and cognitive load was something we could deal with." P12, Senior Vice-President of SE [ATDI-Q9]

By considering the prior framework ATD items, we can observe how the adoption of a certain framework, the choice of utilizing a (potentially unfamiliar) framework over one developed in-house, and the level of embedment of a framework within a software-intensive system constitute an *act of balance*. In fact, the design decision behind the introduction of such items are not *per se* suboptimal, but can nevertheless lead to ATD if the context of a software-intensive system is not correctly interpreted, or if the trade-off analysis such decisions entail are not analyzed with the required care.

Framework not Up to Date. As emerging from discussions in both focus groups, ATD can manifest itself in the form of frameworks present in a software-intensive system which are not up to date. This type of ATD item, referred in academic literature as "technical lag" [155], arises when new versions of the used framework are released, but its update in the system is delayed while continuing development activities on an outdated version. The repayment of this ATD item is often delayed until the framework inevitably needs to be updated, or it is changed in its entirety (e.g., due to the deprecation of a certain version/framework). As noted by P19, this ATD item is often incurred deliberately, as its consequences are only seldom understood in their entirety.

"You see it [ATD] coming. You could change it [framework version] before. But "it's still working... why should I update my dependencies?"" P19, IT Architect [ATDI-10]

As noted in the second focus group, a prominent example of not-updated frameworks are user interface frameworks, which can lead to serious and widespread consequences if their update is consistently neglected in time.

4.3.1.2 Process ATD items

Complex Problem, Simplistic Solution. Underestimating the problem at hand, and adopting a simplistic solution to address it, can lead to the implementation of architectural components which not only are inadequate to support future requirements, but are in some cases even unfitted to properly satisfy the current ones. Such solutions, often caused by time pressure, are in many cases swiftly replaced, making the initial investment required to implement them almost vain. P7 described the presence of this item as follows:

"Changing the new component can be quite challenging, we always have to make sure we don't end up breaking all the edge cases which are not considered. It can be quite brittle and so like I said, we can end up kind of fighting with it. It might be difficult to evolve it to suit our needs." P7, Senior Software Engineer [ATDI-Q11]

In a way, the forces leading to this ATD item can be considered as opposite to those leading to the *Superfluous framework* ATD item. Indeed, the former is rooted in the underestimation of the problem at hand, whereas the latter is rooted in the anticipation of a degree of complexity which is never needed.

An example of *complex problem, simplistic solution* was provided by P7, while describing the integration of a test suite into a software development kit. The test suite was intended to test the interface specifications of all new components of a certain type, referred to as "connectors". Nevertheless, as this connectors varied greatly in terms of functionality, the test suite developers had to adhere to resulted to be a futile exercise. In fact, a large number of corner cases were not considered in the test suite, and developers discovered to *"actively fighting it [test suite]"*, trying to adhere to the generic test suite specifications, while implementing the functionalities characterizing the components.

New Context, Old Architecture. Another ATD item that emerged in our theory regards not paying continuous effort in keeping the architecture of a software-intensive

system aligned with its context, leading to an outdated architecture. P12 argued:

"If you do not adapt your architecture over time, that's when you end up with a big lump of problems. That's were maybe we took too long, 3-4 years passed before we decided that we had to take the time to fix it [architecture]. And that's a huge undertaking." P12, R&D Director [ATDI-Q12]

Participants mostly reported to incur in this ATD item inadvertently. Nevertheless, this item can also be established deliberately, *e.g.,* if driven by a business strategy:

"*The business was to keep the costs down and make as much profit as possible, and after 8-10 years, the architecture was seriously showing its age . . . "* P11, Senior Director of Technology [ATDI-Q13]

By considering the example regarding SQL provided for the framework lock-in ATD item, we can notice how locking-in a specific framework can make a software-intensive system difficult to evolve, leading to an architecture which cannot keep the pace with its evolving context.

In this study, we noticed that the time required for an architecture to become misaligned with its context varies greatly according to the specific case considered, as it depends on the pace at which the context of a software-intensive system evolves. For example, a software-intensive system developed for the banking domain [156], may need to evolve at a much lower pace than mobile apps, which are generally characterized by a rapidly changing ecosystem [53].

The Minimum Viable Product (MVP) that Stuck. A particular instance of *new context, old architecture* emerging in our theory is an MVP that, while intended as a temporary "bare-bones" solution, evolved into the architectural foundation of a system, without properly considering the architectural implications of adopting an immature artifact as architectural basis. This ATD item often happens in start-up environments, or during the implementation of a new architectural component, and is often related to time pressure, lack of architectural awareness, and uncontrolled software evolution:

"It was an MVP solution that is still in place. And we were constantly broadening the scope of the problem. So there was no longer time to pay attention to the MVP, because not only the customers had their defects, but we had also to constantly implement new functionality. So for quite a long time, we just kept adding new functionality, and this problem was never solved." P6, Senior Software Engineer [ATDI-Q14]

Examples of *MVP that stuck* provided by participants were prototypes of a new architecture, immature R&D components, and experimental development branches, which were adopted (deliberately or inadvertently) as architectural foundations of a software-intensive system.

4.3.1.3 Implementation ATD items

Segment of code affected by TD. Rather than originating from a single, important, architectural design decision, ATD can arise from small details regarding the implementation of architectural components, and the relations between them which, by accumulating and worsening over time, deteriorate the architecture of a software-intensive systems. This type of item often manifests itself as dependency issues, such as architectural tangles, poor separation of concerns, and/or tightly coupled architectural components. As described by P13, due to the reach of this type of items, it might be difficult to locate their exact root cause:

"You would say: "Oh, we know what is wrong with this functionality, it's in this one place", but then there is also this other five places that you have to touch, and you end up not really knowing where the problem is" P13, Senior Software Engineer [ATDI-Q15]

Relating to this ATD item, numerous participants mentioned an "architectural debt halo", i.e. a portion of the architecture with hard to define boundaries, where hard to locate debt resides. In P2 words:

"It takes some awareness to understand you are going down a rabbit hole. But when you realize it, you can just change a bit in the periphery, what you can see, you fix a bit of the halo of badness." P2, Software Staff Engineer [ATDI-Q16]

Prominent examples of this ATD item mentioned by participants were architectural components implemented under par, *e.g.*, characterized by unsound use of access modifiers, ambiguous naming conventions, high cyclomatic complexity, and high cognitive complexity.

The Workaround that Stayed. ATD can be introduced in a software-intensive system as a temporary workaround, implemented to bypass some architectural constraints, which over time becomes deeply embedded into the architecture. As described by P8 in [R-Q1], such workarounds can be brought in deliberately, for the sake of development velocity, or triggered by unexpected context changes. Nevertheless, the awareness of the progressive consolidation of the workaround into the architecture

can be inadvertent:

"somehow we ended up with three pathways through the code, first we had one, then two, and so on . . . there was duplication among the three, but also separate pieces to each one, that stuff was not isolated nicely . . . " P13, Senior Software Engineer [ATDI-Q17]

Consolidated workarounds can become so embedded into an architecture that, while their consequences can be evident, it is no more worthwhile fixing them:

". . . at this point . . . I think it's been deemed too expensive at best to change that [workaround], relative to the other business priorities we have." P7, Senior Software Engineer [ATDI-Q18]

Trial-and-error Design. If an insufficient amount of resources is invested in carefully designing a component, an implementation of it can be established by iteratively fixing a suboptimal version of it. This type of ATD item manifests itself as a component (or a set thereof) which has to be continuously adapted in order to satisfy the current and new requirements. P2 describes:

"Trying out and then looking back at the component, you can immediately see if something isn't right, for example if you have a lot of code to do something that should be simple. And then we can say, we passed the bad, I understand now what I have to do in version 2. But then the process repeats itself." P2, Software Staff Engineer [ATDI-Q19]

The example considered in the previous quote regarded an interface of an embedded system, enabling a software component to communicate with its underlying hardware. While similar interfaces were developed in the past, in order to discard legacy implementations, a new interface was developed from scratch. Such interface, retouched multiple times as old requirements were rediscovered, resulted in a trial-and-error design, accommodating requirements incrementally, without any structured upfront design.

4.3.2 Causes

In this section we present the root causes of ATD items emerging from our data. Specifically, we identified two separate type of causes mentioned by the participants, namely *external* and *internal* causes. External causes regard the influence of the

context of software-intensive systems on their ATD. Internal causes instead embody factors inherent to the development and maintenance of the system. As noted during both focus groups, an external cause often leads to one or more internal causes, *i.e.*, a stimulus provided to a software-intensive system in the form of an external cause, may trigger one or more causes internally. An overview of the ATD causes emerging in our theory is depicted in Figure 4.8.

4.3.2.1 External Causes

Time Pressure. 16 of the 18 participants acknowledged time pressure as the main cause of ATD. P11 summarized the concept of time pressure of software development, which most of the participants reported, as follows:

"In a product you need to hit quarterly targets and what not, always be on the treadmill, getting things done." P11, Senior Director of Technology [CA-Q1]

P10 further details this concept by talking explicitly about the relation between ATD and time pressure:

"*The cause [of ATD] is the same as usual, save time!*" P10, Senior Software Engineer [CA-Q2]

As can be evinced also from [R-Q1], under time pressure, architectural quality is often sacrificed. This is a recurrent theme across all participants. As P2 noted:

"*When time becomes tight, the first thing that will fall out is cleaning up the architecture."* P2, Software Staff Engineer [CA-Q3]

The rationale behind the sacrifice of architectural quality for the sake of velocity, has to be attributed to the large amount of resources often involved in architectural changes. This concept is described by P13 as follows:

"*One thing is always time, it's quicker to do feature development instead of doing architectural changes*", P13 - Senior Software Engineer [CA-Q4]

From our data we observe that developers often take architectural shortcuts and accumulate ATD when the time pressure is high, under the (often incorrect) assumption that these shortcomings will be dealt with at a later stage, as further detailed in Sections 4.3.8 and 4.3.9.

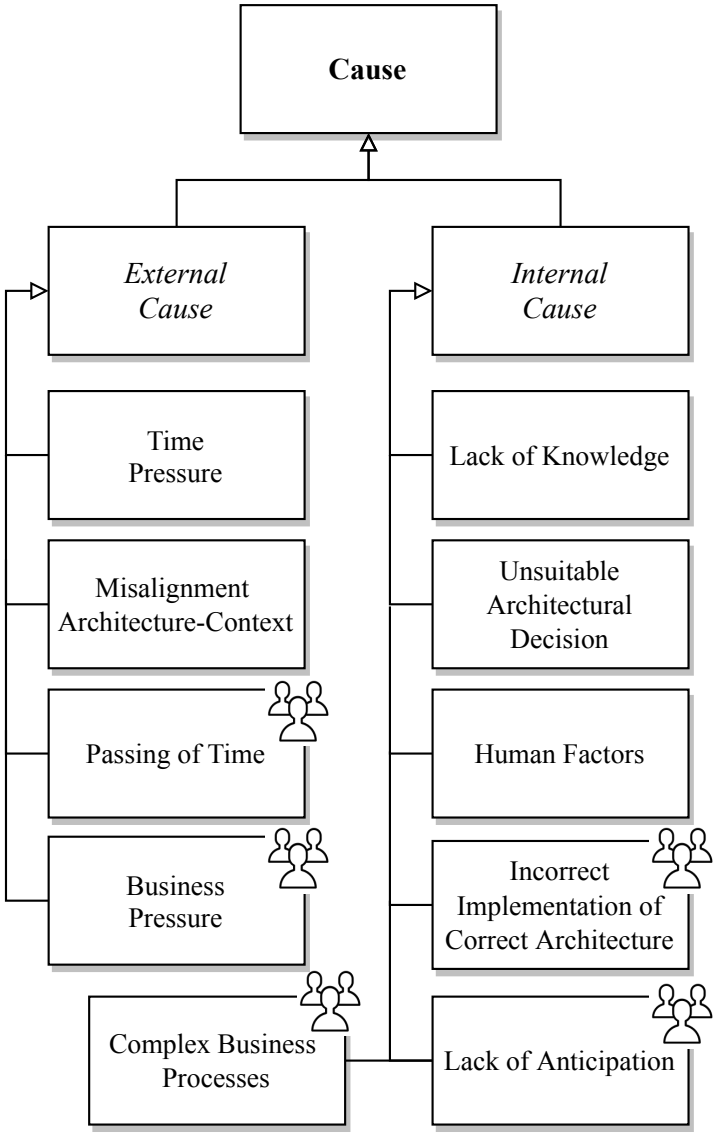


Figure 4.8 – Overview of ATD causes

Misalignment Context-Decision. If the context of a software-intensive system is not clearly understood, suboptimal architectural decisions can be taken inadvertently. Such decisions might lead to the evolution of architectures, not by considering their real context, but a hypothetical, building on the existing debt. P8 recalled one of such instances:

"The abstractions we used didn't really match reality. We thought to know how things had to be done, but thinking back at it... we were completely off!", P8 - Senior Software Engineer [CA-Q5]

Clearly understanding the context of a software-intensive systems results to be one of the paramount factors to mitigate the establishment of architectural debt items:

"It's about the semantics: only if you really know what your concepts are you can create a good architecture. That's why I think the architecture should deeply reflect the requirements, the semantics, the domain that your software system supports.", P17 - Co-Founder [CA-Q6]

Passing of time. Even if the context of a software-intensive system is well understood and all correct architectural decisions are made, passing of time will naturally and unavoidably build up ATD. As noted in both focus groups, *"aging technologies"* is the most common, and inevitable, manifestation of passing of time as a cause of ATD. A participant of the second focus group explained:

"Regardless of the effort spent in maintenance, systems are aging. Even if you consider all potential concerns, the environment changes, and some of the correct architectural decisions you took in the past are just not valid anymore." P25 - Enterprise Architect [CA-Q7]

The *passing of time* cause is related in our theory to specific types of *ATD items*, namely *new context, old architecture*, and *not updated framework*: as components of a software-intensive system slowly become outdated, the architecture further and further gets misaligned with its context, till a maintenance effort is required to pay back the "naturally" accumulated debt, e.g., by changing a certain architectural component, or upgrading a framework to its latest version.

Business pressure. In order to meet requirements of stakeholders, or fulfill commitments taken with them, architectural decisions can be taken, even if the decisions entail undertaking a considerable amount of ATD. Such type of tactical decisions, often taken by business departments, prioritize the achievement of a goal over the

potential consequences in a software-intensive system, either because the consequences are not well understood, or no other option is available. P23 described:

"Business owners do not know how to develop software properly, they push certain decisions because they have made promises and committed on a result, they want to get there, even by making compromise because the route to do it nicely is not possible... and this choices create lots of debt, as they do not mind how it is designed." P23 - ICT Business Manager [CA-Q8]

4.3.2.2 Internal Causes

Lack of Knowledge. In the presence of an unclear architecture, developers often introduce ATD (either inadvertently or deliberately), in order to save the time that should be invested in understanding comprehensively the architectural details.

This situation, often embodied as a lack of, or disorganized, architectural documentation, was described by many participants, including P12, who explained:

"When you are working on an older system, you have lots of constraints that you have to know about, and they are often not well documented, and so you don't know what things will come in your way, things that you have to work around. You are constantly extinguishing little fires to figure out what is going on, it takes a while ..." P12, R&D Director [CA-Q9]

In addition to the introduction of ATD, lack of architectural knowledge can also lead to the obfuscation of ATD items, hence hindering the awareness of the ATD present in a software-intensive system. P2 describes:

"There was no documentation or tests. You never really understood if the code was intended like that, if it was intended that way, or if it was just "I will get to this later"." P2, Staff Software Engineer [CA-Q10]

In both focus groups, participants highlighted that the lack of knowledge leading to ATD does not have to be strictly architectural: lack of context knowledge, standards, technology availability, and company-wide progress awareness, are all instances of lack of knowledge that may cause ATD.

Unsuitable Architectural Decision. ATD can arise by making inadvertently an inappropriate, sub-optimal architectural decision. Often, inadvertent design decisions

leading to ATD are associated to the lack of context awareness, which result in approximate and/or ill-calibrated trade-off analyses. P14 described one of such instances:

"At the time there were reasons that supported our decision, but later on... when we think back at it, we see that we didn't evaluate all options." P14, Senior R&D Manager [CA-Q11]

The magnitude of the ATD associated to unfitted decisions varied greatly across participants, with some notable cases where the impact on the success of a software product was enormous:

"Making that decision didn't seem important at the time, but we should have considered the debt associated to it early on. For me, it was a lack in understanding properly the context... the project eventually got killed." P14, Senior Software Architect [CA-Q12]

Human Influence. A recurrent cause of ATD is the influence of human factors on ATD. Under this category fall aspects related to personal drive, such as the example reported in [ATDI-Q6] (including lack of developer expertise) and cognitive biases (notably the Dunning-Kruger effect [157]). Due to the importance of this topic in our theory, we further discuss findings related to human factors in Section 4.3.9.

Incorrect Implementation of Correct Architecture. From both focus groups emerged that, when an architectural design decision is not *per se* a direct cause of ATD, it is still possible to incur in ATD if such design decision is not implemented correctly. The consequences associated to this type of cause are often of severe nature, as the divergence between designed and implemented architecture leads to an unforeseen state of the system, undermining the tradeoffs considered when the design decision was made. P25 concisely stated:

"You can have a brilliant idea, but if it is not implemented correctly, it can be just debt." P23, Enterprise Architect [CA-Q13]

Lack of Anticipation. Software-intensive systems need to continuously evolve in order to be aligned with their ever-changing contexts. If an insufficient amount of effort is spent in understanding how a software system may need to be adapted in the future, even an architecture which is well-fitted for its current context, may lead to steep ATD as the architecture is required to evolve. As discussed in the first focus group, characterizing, examining, and documenting anticipation can be an exceptionally hard problem, as understanding the amount of required anticipation is

not possible. In P22 words:

"This one [decision] is hard to take. How much anticipation? How much in the future you want to try to look?" P22, Vice-President [CA-Q14]

According to the participants of both focus groups, ATD introduced due to the lack of anticipation is often more evident in organization where Agile development practices are in place, as not many architectural design choices appear to be thoroughly discussed and analyzed with the required depth.

Complex Business Processes. In some cases, complex business processes in place at a company are translated into an architectural complexity of their software-intensive systems, leading to ATD. This instantiation of Conway's law [158] can usually be addressed, rather than by software refactoring activities, only by reviewing the business processes in place in a company, in order to mitigate the potential port of business complexity into the software-intensive system. As noted by the participants of the second focus group, complex business processes can also slow down the maintainability and evolvability of a software-intensive system, by burdening development activities with "bureaucratic" procedures of unclear added value.

4.3.3 Consequences

In this section we document the consequences of ATD emerging from our data. Specifically, we identified consequences of 3 different types, namely *business-*, *functionality-*, and *product-development*-related. In Figure 4.9 an overview of the emerging ATD consequences, and their associated type, is depicted. As discussed by the participants of both focus groups, ATD consequences may take a long time before they become tangible, incrementally worsening till they become visible.

4.3.3.1 Business-related Consequences

Carrying Cost. Often, the consequences of ATD are not immediate, but rather manifest themselves over time. Specifically, a recurrent consequence of ATD is an incremental amount of resources which have to be dedicated over time in maintaining and evolving software-intensive systems. As P1 described:

"We did not think hard enough of the [architectural] design, its cognitive overload, the associated carrying costs, how much will take us on a continuous basis to work on the

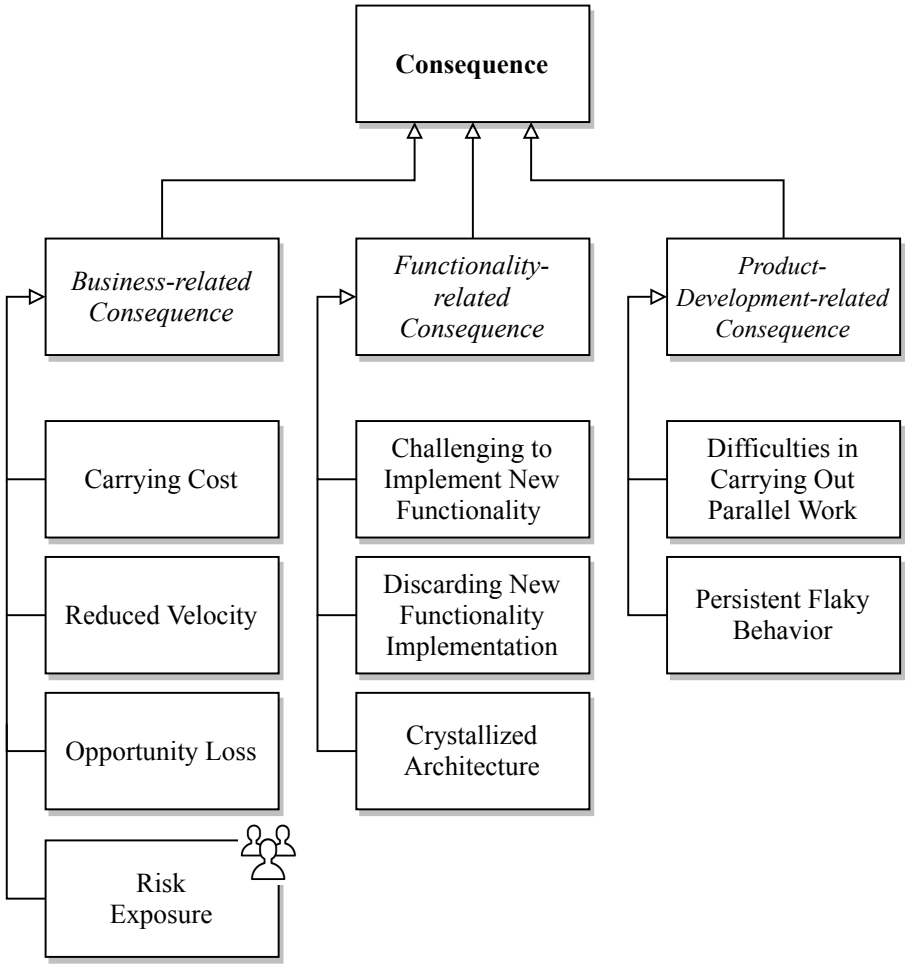


Figure 4.9 – Overview of ATD consequences

system designed this way." P1, Senior Vice President of SE [CO-Q1]

The carrying cost associated to ATD afflicts a software product by requiring an increasing amount of resources for development activities, often imperceptible to end-users, that could be allocated to other tasks. In order to mitigate the negative impact that the carrying cost can have on customer perception, some participants reported to actively invest resources to make refactoring efforts tangible to their end-users:

"While doing the refactoring, we also enhanced the front-end, just to let the customer feel that the product is getting better." P4, Chief Technology Officer [CO-Q2]

Reduced Development Velocity. Related to the first two emerging consequences, most participants described one of the main consequences of ATD as a distinct loss of development velocity. This loss is in most cases associated to additional time required to understand the architecture, modify multiple components when carrying out small changes, and fixing bugs which, due to ATD, are hard to locate. P13 explained:

"Development takes much more time than expected, sometimes because you run into an unknown issue, and other times you just cannot properly size the thing that you are working on, because the architecture is much more complex then what you expected." P13, Senior Software Engineer [CO-Q3]

Opportunity Loss. Due to ATD, opportunities to follow new business avenues can be lost due to the inability the system in order to accomodate them. P3 described:

"You have to go overtime, make changes, and that's where the real cost is, because you spend the time trying to fix those architectural problems, and spending less time in innovation. People pay for something, and they expect it to work." P3, Senior Director of SE [CO-Q4]

The loss of opportunity is proportional to the effort required for ATD management (see Section 4.3.5). While in the previous incident only part of the resources available were dedicated to manage ATD, more drastic strategies, such as a *major refactoring*, can lead to more severe opportunity losses. P17 recalled one of such instances:

"We lost many months on this [ATD], because there was not added value from a functional point of view. We sacrificed implementing new functionality for refactoring. We did not loose customers, but it took more than 6 months to refactor everything." P17, Co-Founder [CO-Q5]

In order to avoid opportunity loss, it is even possible to deliberately postpone the

repayment of debt, and continue to accumulate it till a reactive management strategy is required. P11 explained:

"We had architectural issues, but we had customers, sales, commitments that we had to meet. If we stepped away from that, dedicating half of the team to refactoring, we would not be able to take the new opportunities that came through." P11, Senior Director of Technology [CO-Q6]

Risk Exposure. From both focus groups emerged that a prominent consequence of ATD is exposure to risk. Rather than an ATD consequence which is currently present and impacting a software-intensive system, risk exposure is a *potential* consequence, which may or may not lead to other consequences according to the future evolution of the software-intensive system and its context. Incurring ATD, and the *passing of time* cause, entail a higher exposure to risk, *i.e.*, a higher probability that consequences may occur. The *exposure to risk* cause can be subdivided into two separate variables, namely *probability of consequence*, and *impact of consequence*, both of which are heavily influenced by ATD and the passing of time. As explained by P22 in the first focus group:

"Risk exposure is a mathematical formula: it [the risk] is the probability of something failing, multiplied by the impact when it fails. " P22, Vice-President [CO-Q7]

As observed in the first focus group, while risk exposure is strongly related to business-related consequences, such consequence can be also seen as crosscutting, i.e., as an intermediate level between the *consequence* category and its three associated types. While this consideration stands true in our theory, for the sake of readability, we opted to relate risk exposure to its closest type, namely *business-related consequence*, rather than introducing an additional abstraction level in the theory.

4.3.3.2 Functionality-related Consequences.

Implementing new functionality becomes challenging. Associated to the carrying cost, ATD also can affect the effort required to implement new functionalities. This is often associated with "blurred" responsibilities among architectural components. P13 describes:

"Adding new functionality was more difficult, because we had all these little pieces: it was difficult to figure out what they did, and what needed to be done to add a new feature." P13, Senior Software Engineer [CO-Q8]

Difficulties related to the implementation of new functionalities can make it harder to meet the requirements of the stakeholders, leading to more severe consequences, such as the postponement of planned releases. As described by P15:

"We never met a release plan, we often postponed releases. Few few days before releasing, I asked stakeholders if we wanted to go live. And it was a bad idea to do so, we were still bug fixing. But I did not speak out. The stakeholders had to see it as well, that the debt was hurting them." P15, Chief Software Architect [CO-Q9]

Discarding New Functionality Implementation. ATD can seriously affect the ability to implement a new functionality, to the point that it becomes necessary to completely discard the related implementation. Especially telling are instances in which participants recalled the need to implement a trivial functionality, which was discarded due to ATD. One of such instances was described by P6, who recalled:

"The new functionality, if you talked about it, was so reasonable to do... but in reality... it was so difficult to implement in the current architecture that we ended up scooping it out." P6, Senior Software Engineer [CO-Q10]

Crystallized Architecture. In the most severe cases, the architecture can become "crystallized", *i.e.,* the ATD of a software-intensive system hinders almost completely the ability to implement new functionalities. One of this rare occurrences was described by P4:

"They [software developers] could not even build new features, because of the architectural debt they were facing. They put workaround on workaround, and then they couldn't implement new features, because of this pile of garbage that they built..." P4, Chief Technology Officer [CO-Q11]

4.3.3.3 Product-development-related Consequences

Difficulties in Carrying Out Parallel Work. Due to poor separation of concerns and tight coupling among architectural components, ATD can impact also the ability to carry out parallel development across different teams. This is often occurring in the presence of architectural anti-patterns such as blob components, *i.e.,* components encapsulating a big portion of the business logic or data of a software intensive-system [159]. P14 describes one of such incidents as follows:

"The module became so popular that we just kept building more features on it ... and

now it starts to become a bottleneck, because we have so many teams working on the same code at the same time, that people start to step on one another toes." P14, Senior R&D Manager [CO-Q12]

P1, P14, P5, and P7 recognized that this is due to the cross-cutting nature of software architecture, especially if the concerns are poorly separated among architectural components. P1 argued:

"If you cross the boundary and have to touch the architecture, a lot of what is built on it will change. If the modules are not well isolated, who's working on them will be hold at bay. You have to say: "No, you're locked!"" P1, Senior Vice-President of SE [CO-Q13]

Persistent Flaky Behaviour. Software-intensive systems afflicted by a severe amount of ATD can become unpredictable in terms of expected behaviour. This dreadful state of a system is in most of the cases co-occurrent with a *crystallized architecture*. Since in those cases the ATD item causing the issue is often impossible to pinpoint, a *rewrite from scratch* of the whole system is often the only viable solution (see Section 4.3.5). P8 recalled:

"We had to rewrite an entire server side application for a capital market trading app, it was just randomly crashing. JVM out of memory, synchronized deadlocks, like every Java nightmare scenario possible. It was a nightmare." P8, Senior Software Engineer [CO-Q14]

4.3.4 Symptoms

An overview of the ATD symptoms is presented in Figure 4.10. All participants described symptoms which point to ATD items. This led to the emergence of four different types of ATD symptoms in our theory, namely symptoms related to *issues*, *resources*, *performance*, and *development practices*. Similar to the medical domain, symptoms can point to the potential presence of ATD in a software-intensive system, especially if multiple symptoms co-occur at the same time. Symptoms are linked to consequences, specifically, they are consequences that are observable. In contrast, not all consequences are visible, and they may have different granularity: some could manifest themselves at the level of an individual ATD item, while some other at the level of the whole system.

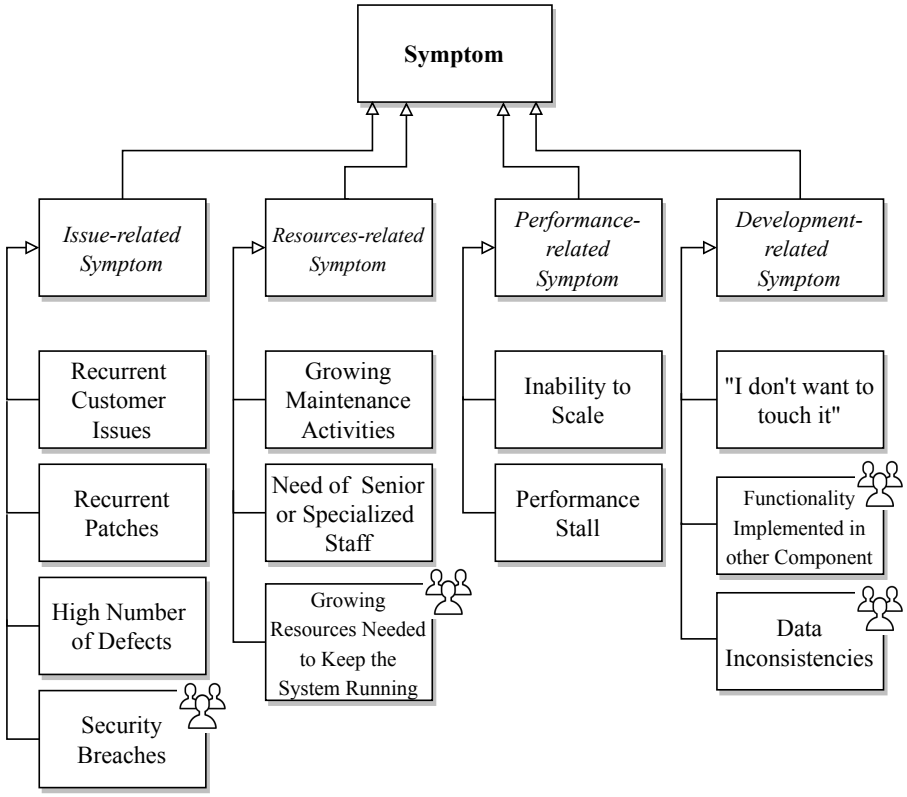


Figure 4.10 – Overview of ATD symptoms

4.3.4.1 Issue-related Symptom

Recurrent Customer Issues. Among all symptoms of ATD, recurring customer issues is the most apparent one. As P3 explains:

"The best indicator of all are customer issues: if you have an area with lots of recurring customer issues, either the team is garbage, or you have architectural issues." P3, Senior Director of SE [S-Q1].

In addition to helping to localize ATD problems in software-intensive systems, recurrent customer issues also guide the timing to start refactoring activities. P4 recalled:

"When we decided to refactor the architecture? It was just the number of customer issues. Who does not have them? But when we started seeing that the spike in customer issues was going to affect our growth, it became something that we had to address." P4, Chief technology Officer [S-Q2]

Recurrent Patches. Linked to the previous symptom, the presence of an ATD can be identified by observing which portions of a software architecture are patched frequently. As the recurrence of patches in an area of code is often not kept track of, numerous iterations may be necessary before an ATD item is uncovered. In P9 words:

"There's this kind of hard to pin down feeling, when in order to meet some new need you are like "okay, it feels weird but I'll patch it, and I'll patch it again, and again, and again. And after a while, you realize that you're kind of like, always applying kind of... you're playing whack-a-mole! It can't be that everything is an edge case!"" P9, Vice President of Product [S-Q3]

High Number of Defects. As reported by many participants, a high number of defects localized in a certain area of the code can indicate the presence of an ATD item. In this context, we refer to *defect* as a generic problem in the source code of the system, such as a bug, a security vulnerability, etc. As P13 explained:

"When you have a lot of bugs in an area of code, that means: either that area is complex by itself, or there is some unmanaged architectural complexity leading to that." S10, Senior Software Engineer [S-Q4]

As for the previous symptom, data regarding defect density and recurrence is not often systematically stored and analyzed to optimize software architecting and development processes. This leads to rely on the experience of senior practitioners,

in order to intuitively detect the emergence of ATD items by considering this symptom. As described by P10:

"Where to fix the architecture is usually decided by experienced people observing that this area creates a lot of defects over the last couple of months, and we need to look at it sooner or later." P10, Senior Software Engineer [S-Q5]

Security Breaches. Security breaches are a recurrent symptom of ATD. Due to the complexity caused by the ATD present in a software-intensive system, inadvertent security flaws can be introduced, leading to the unintentional disclosure of private information to unauthorized parties. Such data leaks can be a strong signal of ATD, that has to be tackled with a reactive management strategy (see Section 4.3.5) as soon as the symptom arises. Due to the sensible nature of the subject, participants could not provide concrete examples of the occurrence of security breaches; nevertheless, it was recognized as a prominent symptom of ATD in both focus groups.

4.3.4.2 Resources-related Symptoms

Growing Maintenance Activities. The presence of prominent ATD items can be noticed by the need to allocate a growing amount of resources without observing a noticeable increase in productivity. P3 summarily described:

"As we added more and more developers, we were not adding many features, why? Productivity, usability, all those things were not in the architecture." P3, Senior Director of SE [S-Q6]

In addition to the growing effort needed to implement new features, concerning amounts of ATD can be noticed by the need of allocating dedicated teams to maintenance and refactoring activities. This results to be a common practice which, due to the severity highlighted by this symptom, is often followed by a *major refactoring* or a *rewrite from scratch* (cfr. Section 4.3.5.2). An occurrence of this ATD symptom was described by P9 as follows:

"We basically had to subdivide our hub team into two, one team dealing only with bugs, and one dealing with features. It was brutal." P9, Vice-President of Product [S-Q7]

Need of Senior or Specialized Staff. Due to the complexity that ATD items entail, their presence can be noticed by the growing need to on-board senior staff into development teams. As discussed by P11:

"You notice it [ATD] by the increasing need to bring in senior people. Because that means that there is something that requires deep, profound understanding. And if there is a major shortcoming, you may have to know something very very deep in order to see it. That usually hints at an emergent area that you will need to tackle." P11, Senior Director of Technology [S-Q8]

Related to the *person* and *communication* categories of our theory (see Section 4.3.9 and Section 4.3.10), seniority is also required in order to effectively *expose* the presence of ATD items. In most of the incidents recalled by practitioners, only senior staff possessed the knowledge and confidence necessary to openly discuss and address ATD. P2 shared his personal experience on this:

"As long as I was junior, I could not say "Hey, this architectural pattern sucks, let's do something about it". I was more quiet. When I was able to have a louder voice... it all started with being noisy and seeing what senior people did to clean up." P2, Software Staff Engineer [S-Q9]

As noted in both focus groups, in addition to senior staff, this symptom may manifest itself also in the need to on-board staff with a particular set of skills. Such specialized staff, often possessing "outdated" skills, may point to the need of modernization of a software-intensive system, and constitute a contingent liability due to the scarce availability of such skill in the current job market. Participants of the first focus group agreed on the need of programmers familiar with COBOL, a language first appeared in 1959 and still widely adopted in the business sector [160], as a prominent example of the *need of specialized staff* symptom.

Growing Resources Needed to Keep the System Running. As noted in the second focus group, a symptom of the presence of ATD is a growing number of resources required to keep the system running. Rather than resources needed to evolve or maintain a software-intensive system, this symptom embodies a continuous amount of resources that have to be allocated to *sustain* the system. Resources associated to this symptom can be both of monetary nature (*e.g.,* cloud provider commissions), or manual effort (*e.g.,* manual interventions required to handle corner cases). An example of this type of symptom was described by P19, who recalled:

"Due to our design, we needed to use a hybrid cloud model. And this [decision] caused a lot of network traffic. And, as you need to pay for network traffic, accounting started to tell us 'Hey, how come you are spending so much money now?' " P19, IT Architect [S-Q10]

4.3.4.3 Performance-related Symptoms

Performance issues which are hard to address can also be a symptom of ATD. From our data, two types of performance issues emerged, namely *inability to scale* and *performance stalls.* P3 illustrated this symptom as follows:

"You can feel it [debt] around performance, you can feel that the architecture is not good enough, because you can feel the performance problems that you fix, a lot of those exist because they are not architected well" P3, Senior Director of SE [S-Q11]

Inability to Scale. Inability to scale refer to the presence of scalability issues in software-intensive systems due to ATD-related problems. This is a recurrent symptom among our participants, and is often characterized by a swift increase of data to be processed. P14 recalls:

"One of the biggest architectural problems we had related to architectural debt was dealing with scale. The system could not cope with the new amount of data, it couldn't work with the current state of the architecture." P14, Senior R&D Manager [S-Q12]

Architectural shortcomings that are identified by considering scalability issues often point to debt items which require a considerable effort in order to be fixed, such as the re-implementation of various portions of an architecture. P14 describes:

"We thought "the system is built that way", but at the time we did not think that we had to scale up that much, and we had to rethink stuff, we had to update things to the newer standards." P14, Senior R&D Manager [S-Q13]

Performance stall. Performance stalls indicate performance bottlenecks present in software-intensive systems which cannot be solved without architectural refactoring. P3 described this symptom as follows:

"With performance, if you can really just move it around but not solve it, that is an indicator that you are doing something architecturally wrong." P3, Senior Director of SE [S-Q14]

Performance stalls can lead to the investment of a conspicuous amount resources to carry out small optimization of an architectural deficiency, which in reality can only be with a proper, structural, architectural refactoring. As P14 states:

"Even if you put in a lot of hack to make it faster, you cannot fundamentally make a change, it is an unsolvable problem. If you have unsolvable problems, that's because of

an architectural decision which is just not right." P14, Senior R&D Manager [S-Q15]

4.3.4.4 Development-related Symptom

"I don't want to touch it". This symptom of our theory deals with human intuition and sensitivity. Rather than deriving from a systematic analysis, this symptom represents the instinctual refrain of software developers to modify a certain component in which ATD resides. R12 describes one of such instances, associated with a "dormant" ATD item:

"Developers will often tell you if something stinks, right? There is always something which is hard to work with, maybe it's a piece of code that no-one wants to touch, that's a symptom! Why does no-one want to touch it? Because it's [bad]! It might do its job well, but no one wants to touch it! [...] Developers: they are the best source of truth when it comes to how healthy your code is." P12, R&D Director [S-Q16]

Functionality Implemented in other Component. From both focus groups emerged that the presence of ATD in a component can be noticed if functionalities, which should be implemented in that component, start to appear in other components instead. This may indicate that evolving the component where ATD resides results to be too cumbersome, and hence the implementation of that functionality has to be delegated to one of its surrounding components. As P21 illustrated:

"You see it [ATD] in the compensation by other components. You have a component which is not good enough anymore, and you see functionality appearing in the surrounding components, which are interfaced with that component, but those functionalities just don't belong there." P21, Director Enterprise Architecture [S-Q17]

Data Inconsistencies. As discussed in the first focus group, data inconsistencies is another symptom which can point to the presence of ATD. Specifically, this symptom manifest itself as multiple instances of the same data, stored in different portions of software-intensive system, which are not consistent with one another. Prominently, this symptom arises when organizations merge different software-intensive systems, but do not have the time to carefully design and implement the integration. This leads to the adoption of architectural shortcuts, disregarding to avoid the storage of the redundant yet divergent data, often represented in multiple formats (*e.g.*, dates), in different portions of the system. As an example provided by a P22, when booking an airline ticket upgrade by utilizing reward miles, the loyalty program website may

indicate that the upgrade is confirmed, while the official airline site shows the upgrade status as pending, and it is impossible for the user to find out which status is correct until they board the plane.

4.3.5 Management Strategies

Six managements strategies to cope with ATD emerged from our data. Interestingly, such strategies focus on the management of ATD items, rather than resolving their root causes. By inspecting the ATD causes, we can conjecture that this is due to the generic nature of the causes (with special emphasis on the external ones), leading management strategies to address them to fall out of scope of the theory investigation topic. We identified three types of management strategies, namely *active*, *reactive*, and *passive*. An overview of the strategies is depicted in Figure 4.11, and further described in the remainder of this section.

4.3.5.1 Active management strategies

Active strategies are based on the acknowledgment of the presence of ATD in a software-intensive system, and the development of a plan to actively manage it. In the following we present the three active management strategies emerging from our grounded theory.

Boy Scout Rule. This management strategy is often referred to by our participants as "The Boy Scout Rule", which borrows from the "Always leave the campground cleaner than you found it" camping rule. Based on this metaphor, developers acknowledge the presence of ATD, and pay back the debt in small incremental steps while carrying out other development activities on a software component, such as the implementation of a new functionality or bug fixes. As P1 described:

"I generally advocate in "stealing time", when a component has bothered you enough, I would just say: fix it, and do not tell anyone. If you are already working on that area of code, just take some extra time to refactor it." P1, Senior Vice-President of SE [MS-Q1]

However, it is important to stress that this strategy can be difficult to apply in practice since ATD items are hard to fix in small increments, unlike other forms of TD. For example, the switching towards a different programming language, substituting a third-party component or platform, or refactoring a deeply tangled subsystem can

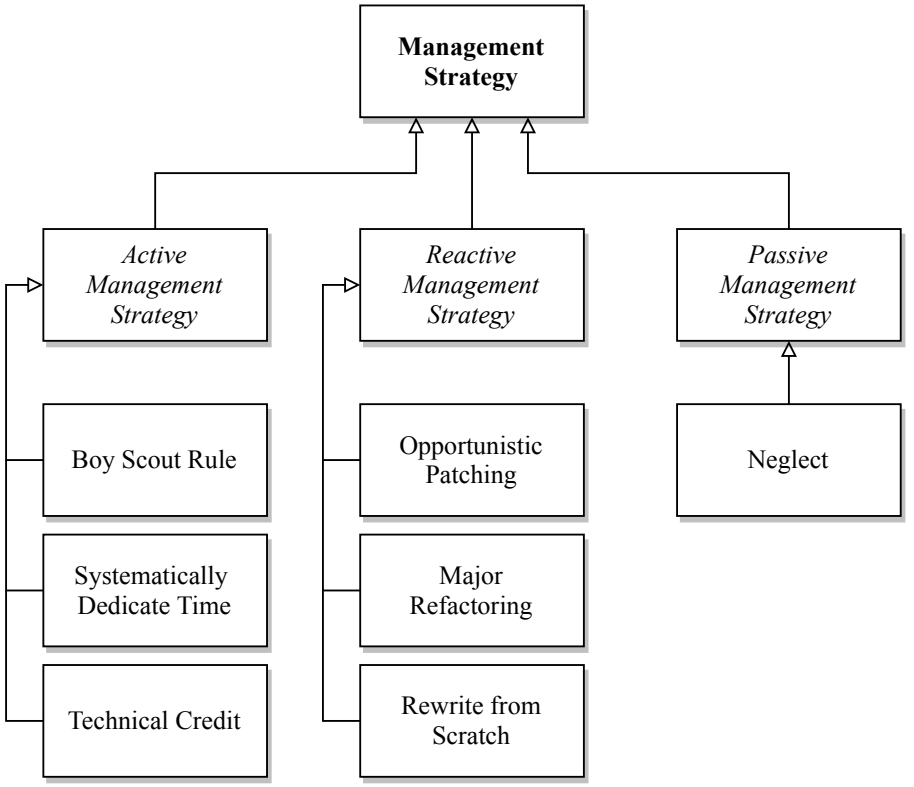


Figure 4.11 – Overview of ATD management strategies

have a pervasive and costly impact on the architecture of the system, potentially requiring considerable effort.

Systematically Dedicate Time This management strategy entails systematically allocating time in order to repay the accumulated ATD. Most participants described allocating a fixed percentage of development time per-sprint to refactor ATD items. The most recurrent percentage of time dedicated to ATD refactoring results to be between 20% and 30%, with the exception of P1 and P9, who reported 10% and 50% respectively. In a singular instance, P12 jokingly described allocating an entire day per-sprint exclusively to ATD refactoring activities:

"We have a Lannister day, you know, because Lannisters always pay their debts [laughs]." P12, R&D Director [MS-Q2]

Technical Credit. This management strategy regards the investment of resources to improve the architectural maintainability and evolvability of a software-intensive system prior to the emergence of ATD items. This strategy aims to mitigate the future establishment of ATD by estimating and proactively addressing portions of the architecture which could slow down future development. While some participants described this strategy from a theoretical standpoint, the common agreement among participants is that, due to time pressure and the uncertain outcome of this strategy, it is hardly ever adopted. P3 explained:

"You are spending time in trying to make something perfect. When do you have that time for that? Where do you take the investment? You do not get paid by "I'll make it evolvable", you spend days or weeks in something that might not pay off, who can afford that?" P3, Senior Director of SE [MS-Q3]

4.3.5.2 Reactive Management Strategies

Reactive strategies entail that, while the presence of ATD in a software-intensive system is acknowledged, its management is postponed until the repayment becomes unavoidable (*e.g.,* an ATD item prevents the development of a new feature). The following reports on the three main reactive strategies emerging from our data.

Opportunistic Patching. This strategy, rather than aiming at resolving the ATD present in a software-intensive system, deals with its occurrence by investing the minimum resources necessary to bypass the limitations imposed by the ATD. This

often results in small patches, or temporary architectural workarounds, which build upon the existing ATD. As described in [S-Q3], opportunistic patching rarely achieves the resolution of the root cause of an ATD item, but can rather point to the underlying problem. A similar situation was described by P11:

"It was architectural debt, but we were able to squeeze around it by doing little incremental changes here and there, which did not touch the architecture much, but slightly improved things... we were just kicking the can down the road... in retrospective we were just patching, patching all the way." P11, Senior Director of Technology [MS-Q4]

Major Refactoring. Due to the severity of the ATD present in a software-intensive system, it can become necessary to methodically eradicate it, even at the cost of sacrificing other development activities. This constitutes a major undertaking, which can cause the loss of competitive advantage of a software-intensive system, and is characterized by investing a conspicuous amount of resources. Many participants referred to this strategy as "biting the bullet", to express the severe influence of this strategy on other development activities. Under this category fall architectural refactoring activities carried out by entire developer teams. Due to the major implication of carrying out major architectural refactoring and the uncertainty of its outcome, timing this strategy can be a complex problem. P11 explains:

"There is always some inertia, you always have to overcome this lump of "when is the right time?", because there is never a right time. You have to decide when it is the right time. Usually it would be based on how painful it is. It has to reach some sort of crest before you realize: "OK this is enough now", you bite the bullet, and try to do something about it..." P11, Senior Director of Technology [MS-Q5]

Rewrite from Scratch. In the most severe cases, the only way to cope with the crippling ATD accumulated in a software-intensive system is declare "technical debt bankruptcy", and conduct a *tabula rasa* re-engineering of a software intensive-system. This process, often referred to by practitioners simply as "rewrite", consists in re-implementing large portions of a software-intensive system without re-using source code, and is conducted by extracting from the old system its functional- and non-functional requirements, and subsequently re-implementing the requirements in a new system. P13 recalls:

"At some point we had to refactor the product, it had architectural issues. There were some big things that we had to fix, and so we had to rewrite the product entirely... we had no other choice!" P13, Senior Software Engineer [MS-Q6]

Rewriting a software product from scratch provides the opportunity not only to pay off in one go all the accumulated ATD, but also to gain technical credit by associating to the rewrite a software modernization process [161], *i.e.,* upgrading the architecture by adopting newer architectural styles, stacks, technological frameworks, etc. In addition, the green-field nature of the rewriting process provides the possibility to get rid of old bad development practices, which potentially led to the establishment of ATD in the first place. As P9 describes:

"I really wanted the product to go faster. And so I said, please choose a different stack, use a different repo, use a different team, so that we don't inherit all that legacy stuff. And so we basically had to stop development in the old way, port all the features over, and build it [the product] on the most new shiny tech that people like." P9, Vice-President of Product [MS-Q7]

While software rewrites can provide exceptional benefits, they also entail a very high risk, as they are characterized by an uncertain outcome, potentially leading to the complete loss of the resources invested in them. P1 clearly explained:

"I really like the rewrite pattern... people are scared by it, but I did seven. You just develop them on the side. They are hard to pull off, but they work great." P1, Senior Vice-President of SE [MS-Q8]

As hinted to in the previous incident, software rewrites are often carried out in parallel to daily development activities, *e.g.,* via a dedicated team. This resulted to be a common practice in the experience of the participants. Nevertheless, in the most extreme cases, product rewrites can require most of the resources available. One of such instances was recalled by P8 as follows:

"It was a six month effort non-stop rewrite. No new features. I saw the entire department go under ... it was just a nightmare." P8, Senior Software Engineer [MS-Q9]

4.3.5.3 Passive Management Strategy

The passive management strategy, rather than aiming to actively pay back ATD, attempts to cope with it by avoiding to address ATD items.

Neglect. Participants described strategies in which, while the negative impact of the ATD residing in their system might be evident, the cost involved in fixing it was not

worth addressing it. In such cases, development activities are carried out at a slower pace, embracing the ATD, and building upon existing debt.

"Sometimes you have a lot of edge cases but you just, you know the cost of... you know it's bad, you know you don't want to do it, you know there's a better way, but the better way isn't worth it." P9, Vice-President of Product [MS-Q10]

As noted by the participants of the second focus group, in specific instances neglect may be a sound strategy to adopt, as the interest of ATD might have to be never paid back, or might be completely amortized by other necessary development activities, *e.g.,* the substitution of an architectural component, that has to be changed for motivations other than the ATD it accumulated.

4.3.6 Tool

In this study, the adoption of tools to explicitly identify and manage ATD did not emerge as an established industrial practice. As described by P10 in [R-Q8], such tools are either unknown by practitioners, or simply unutilized. This resulted to be a recurrent theme across participants. We conjecture that this finding could be caused by either (i) the perceived immaturity of ATD tools, (ii) the perceived usefulness of ATD tools, or (iii) a current knowledge gap between research advancements and industrial practices.

While no ATD tool appears to be actively used, participants mentioned the use of source code quality analyzers and collaborative code review tools, which are often embedded in the development workflows (e.g., via Git pre-commit hooks[6]). Specifically, SonarQube resulted to be the most established tool, while other prominent ones were Clang Tidy, Git Gerrit[7], FindBugs[8], and PyCharm[9]. Associated to such tools are the concepts of: *quality gates*, which are often customized by developer to fit their needs; *warnings*, used to enforce software quality standards of committed code; and, *automated refactorings*, used to automatically fix small software quality shortcomings.

[6] https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks
[7] https://www.gerritcodereview.com/
[8] http://findbugs.sourceforge.net/
[9] https://www.jetbrains.com/pycharm/

4.3.7 Artifact

ATD items can affect and reside in one or more artifacts. Commonly, given the widespread nature of such architectural debt items, numerous artifacts are simultaneously affected by a single item. An overview of the concepts constituting the *artifact* category of our theory is reported in Figure 4.12 and described below.

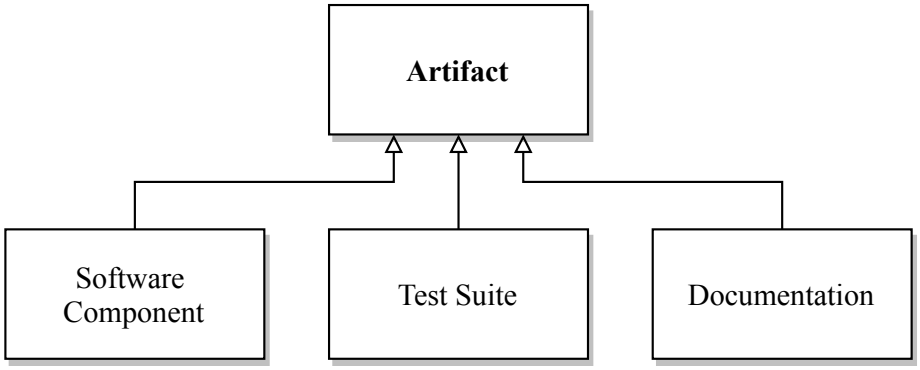


Figure 4.12 – Overview of ATD concepts related to *artifact*

Architectural component. The ever-present artifacts in which ATD items manifests themselves are architectural components. Such portions of the codebase, encapsulating one or more functionalities of a software-intensive system, are in most cases the root location where ATD items are originating. In rare instances, ATD items can also spawn from the relations established between components, *e.g.,* due to debt accumulated in an Application Programming Interface (API), or due to over-complex dependencies. P13 describes:

"We don't even understand the whole code. If some data gets corrupted in that connector, it is hard to tell where the data came from. And that component is very connected to the other ones, and some portions of the code do not follow any pattern. " P13, Senior Software Engineer [A-Q1]

Test Suite. Test suites result to be often affected by debt items residing in architectural components. In fact, the increasing complexity and design issues residing in the architecture of a software-intensive system is frequently reflected in its test suite, which also grows in complexity, loses effectiveness, and becomes harder to maintain

(cfr. [R-Q5]).

Documentation. Architectural debt items can be reflected in a partial, absent, or even erroneous documentation of the architecture of a software intensive-system. Remarkably, this is often due to the growing complexity of an architecture, and/or a loss of overview over the architectural structure of a software-intensive system. Documentation artefact affected by ATD can lead to vicious cycles, in which the resulting documentation debt is both the consequence and the cause of new debt. P17 described:

"There is no documentation... when someone new comes on the team we have to explain the whole architecture, but are we always doing it right?" P17, Co-Founder [A-Q2]

In a peculiar case recalled by P9, the documentation of a software product itself, which reached further away than controllable, hindered the evolvability of a software architecture.

We are kind of fighting against our own success. There are hundreds of tutorials, which would now be wrong. And so we have this sort of like mass of backwards compatibility that allows some changes to be made and other that don't. P9, Vice-President of Product [A-Q3]

4.3.8 Prioritization Strategies

The following discusses our findings related to how the refactoring of ATD items is prioritized with respect to other development activities, such as feature development and bug fixes. Prioritization strategies can guide management strategies of active nature, as reactive and passive strategies respectively manage ATD only when strictly necessary and not at all.

From our results emerged that often ATD is kept track of, *e.g.,* by characterizing backlog items according to the classification of Kruchten [154], who makes the distinction between functional features, bug fixes, architectural features, and technical debt. Nevertheless, while ATD items are often traced, prioritizing their refactoring with respect to to other development activities does not follow an established methodology. As P10 states:

"We fear we do not have a scientific method here... it is basically gut feeling. We do not have any research around what needs to have the highest priority." P10, Senior

Software Engineering [PR-Q1]

This "gut feeling" is a recurrent theme among participants on how ATD items are prioritized. Due to the difficulties associated with quantifying the impact of ATD, practitioners do not adopt systematic prioritization approaches; rather, they adopt informal ones, to balance their ATD refactoring activities with other development activities, as reported also in [R-Q7]. P3 further clarifies this concept:

"I would say, find your balance, do the minimum necessary. It is not a science, I think it's an art. And why do large companies fail? Because at some point that balance is tilted." P3, Senior Director of Software Engineering [PR-Q2]

4.3.9 Person

This category deals with concepts related to the human nature of software professionals. As can be deduced from Sections 4.3.4 and 4.3.8, people can support the discovery and prioritization of ATD items, and are ultimately at the origin and resolution of many of them.An overview of the concepts constituting the *person* category of our theory is reported in Figure 4.13 and described below.

Awareness. To be able to manage ATD one must first be aware of its presence in a software-intensive system. Sharing knowledge about ATD items, their magnitude, causes, and consequences, enables gaining a common understanding of the ATD presence, leading to finer-grained strategies to cope with it. P4 describes:

"What important is the culture of knowing about the debt. We have to be extremely conscious about it. Every developer has to be aware about "I am incurring debt now, I will have to pay this at some point". And this is a very good example of a developer who is aware of it." P4, Chief Technology Officer [PE-Q1]

Personal Drive. Participants often reported *"the personal drive of individuals"* being at the origin of the identification, management, and resolution of ATD items. People championing for a certain ATD item are usually the ones who are affected by it on a daily basis, and actively advocate for its resolution. One of such occurrences is described by P6:

"It comes down to socializing it [ATD item]. You have to be [an] advocate for it. Bring it up in group meetings and one-on-one with certain people. Make sure that they absorb

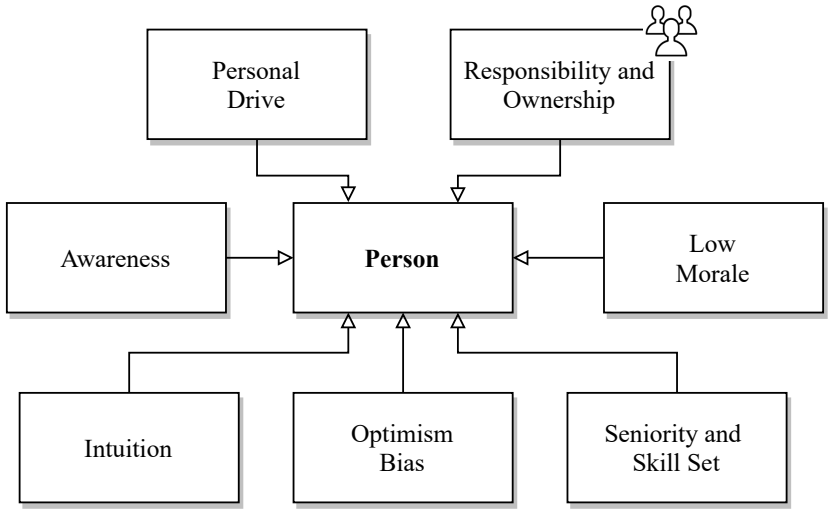


Figure 4.13 – Overview of ATD concepts related to *person*

it, and hold it in the same kind of severity that you do." P6, Senior Software Engineer [PE-Q2]

Low Morale. ATD can have a deep negative effect on developer morale. Due to the encompassing and complex nature which ATD entails, the debt caused by ATD items can affect development activities over a prolonged period of time, leading to severe consequences on the morale of developers. P2 describes:

"*From a human perspective, if you wake up every day and you walk around in mud, are you motivated in doing it? You don't put much effort in it. Which then spirals in not getting much done... and then ends up with people leaving...*" P2, Software Staff Engineer [PE-Q3]

The detachment between personal drive and a software intensive system due to ATD described in [PE-Q3] is further detailed by P15:

"*The people that left before, were just able to cope with the debt, clock out after work, and dealt with the problems the next day without much thought.*" P15, Chief Software Architect [PE-Q4]

Seniority and Skill Set. Seniority and skill set can play a decisive role in ATD related phenomena. On one side, lack of necessary skill sets can lead to the introduction of ATD, due to a lack of fitted resources to address properly an instance at hand. P14 recalls:

"We had experience in monolithic applications, and that's the key reasons why we stayed with this gigantic code base. I wish we could have evaluated other options, but back then nobody in our team had the experience...it's a bit of a pain right now. P14, Senior R&D Manager [PE-Q5]

On the other hand, seniority and adequate skill sets are crucial in order to solve complex ATD items. Participants often described seniority as a decisive factor to address ATD for two main reasons: (i) senior developer are able to gain a better "holistic" view of software-intensive systems, and (ii) junior developer refrain from addressing ATD, due to the magnitude and resonance of changes carried out at the architectural level. P5 explains:

"Junior people don't want to change the architecture. Few people are confident enough to do so, there is a difference between imagining a change and pulling it off, a lot of people shy away from it." P5, Senior Software Engineer [PE-Q6]

Intuition. As described in [S-Q3], [S-Q16] and [PR-Q1], intuition and "gut feelings" can affect ATD by enabling to identify and prioritize ATD items. Additionally, personal intuition is also referenced by our participants as playing a role during the evaluation of the root causes, consequences, and magnitude of ATD items. P7 describes:

"It's a discussion about the gut feeling of how big something is. We don't have any story points associated with them, any well-defined number, it's just based on what each of us knows, what the problems entail, and how we can solve them." P7, Senior Software Engineer [PE-Q7]

Optimism Bias. From our data emerged that inherent optimism of software developers and alike can deeply influence ATD. While optimism is crucial for the success of a software product, it can also constitute a cognitive bias which hinders development activities. P3 explains: *"Everything seems possible! It's just ego. This is always the problem. Think of software development, it's the art of making things possible, right? We can do it, of course we can! How long it will take is a different question... developers have to be optimist, otherwise they don't even start."* P3, Senior Director of SE [PE-Q8]

Our participants reported a wide range of cognitive biases associated to the opti-

mism one, such as wishful thinking, self-serving bias [162], and the Dunning-Kruger effect [157]. Such biases notably lead to the emergence of the planning fallacy phenomenon [163], as described in [PE-Q8]. In addition to planning fallacies, the optimism bias and other related ones can lead also to the introduction of ATD. P6 reports: *"When we made this decision we assumed that, as our interactions were simple, they will continue to be simple. Plus, as they're all SQL databases, we assumed that they're probably pretty similar. So it's very easy to say that, as they are similar, "let's just pretend that they're all the same". And that was just a bit of optimism, but it resulted in many problems."* P6, Senior Software Engineer [PE-Q9]

Responsibility and Ownership. People working on a software-intensive system can be mapped to specific ATD items residing in the system. This type of mapping has a twofold nature. On one side, ATD items can be traced back to the people who intentionally or inadvertently introduced it. On the other side, ATD items can be assigned to specific people who take ownership of those items, and are in charge of managing them. As discussed by the participants of the first focus group, a systematic mapping of ATD items to people can support the management of ATD by distributing responsibilities across development teams.

"If you don't have clear responsibilities and accountabilities, who feels responsible for the ATD?" "... only the seniors ..." P21, Director Enterprise Architecture, and P22, Vice President [PE-Q10]

4.3.10 Communication

We identified 4 main concepts related to the *communication* category, namely *exposition*, *impediments*, *blame*, and *communication with stakeholders*, as depicted in Figure 4.14 and described in the following.

Exposition. Rising awareness among developers, managers, and the like, of the presence of ATD items results to be an important aspect steering ATD management and prioritization strategies. As described in [PE-Q2], pointing out the rise and establishment of ATD items can build a common knowledge among developer teams, leading to a comprehensive and shared viewpoint of the ATD present in a software-intensive system, which could not be established individually. P10 describes: *"Engineers get frustrated that they can't implement functionalities fast enough, so they complain and get vocal about it. This creates a situation where the architectural debt gets more awareness."* P10, Senior Software Engineer [COM-Q1]

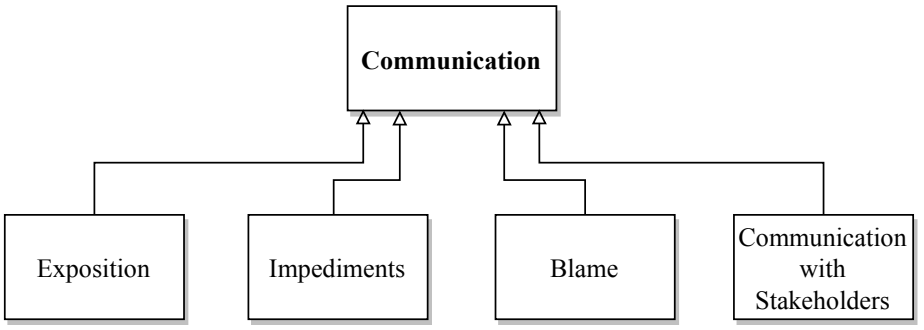


Figure 4.14 – Overview of ATD concepts related to *communication*

Impediments. Related to the communication of ATD, data showed that creating awareness on the severeness of the ATD present in a software-intensive system is not always an easy task. This problem often lies in the communication between developers and management teams, potentially due to unclear consequences and symptoms associated to ATD items. Sometimes this leads to the negation of existing ATD, which can be detrimental to personal drive, and morale of developers. In this regard P8 stated: *"It resembles a Dr. Phil Show intervention. To fix a problem, you have to acknowledge that there is one."* P8, Senior Software Engineer [COM-Q2]

Blame. Incurring ATD inadvertently, or leaving undocumented the rationale behind deliberately incurring it, can lead to friction among people working on a software-intensive system. In fact, without a proper knowledge of the circumstances in which the debt occurred, undesirable discussions can arise, often finger-pointing individuals who incurred in the debt. P4 describes:

"People know when they are incurring architectural debt. And if the people leave, afterwards it's a blame game on who is the culprit. Developers blame the old ones for taking bad architectural technical decisions, because they were not in their position." P4, Chief Technology Officer [COM-Q3]

Communication with Stakeholders. Related to communication of ATD, in our theory emerged difficulties in communicating the presence of ATD to the stakeholders of a software product. As simply expressed by P4:

"People pay for something, they bought it and expect it to work, and then time passes and the product evolves, and course they expect it to work, always, forever!" P4, Chief Technology Officer [COM-Q4]

As ATD accumulates, implementing new functionality becomes more challenging. Similarly, also the issues related to development impediments become more difficult to be discussed with the stakeholders. P3 describes:

"It [the product] should have been maintained without adding new functionalities... hard to communicate that to customers, because they demand "why don't you add more features to it?", but don't know that adding more features takes longer, is harder, causes more problems in an old stack." P3, Senior Director of SE [COM-Q5]

In order to mitigate potential issues related to stakeholder communication, a common strategy adopted is to deliberately spend efforts in making maintenance efforts tangible to stakeholders, giving the impression that the product is still evolving, even when almost exclusively refactoring activities are carried out (cfr. [CO-Q2]).

In extreme cases, the impediments related to communicating ATD to stakeholders can become so prominent, that it may be necessary to deliberately highlight the issues present in a software intensive-system, in order to convince that a major refactoring activity is necessary (cfr. [CO-Q9]).

4.4 Related Work

As recommended by Glaserian GT principles [136], to mitigate confirmation bias, we reviewed the related literature *after* building our theory. From the inspection of the ATD corpus, we identified four studies related the closest to ours [164, 165, 72, 166]. In particular, in these researches we identified a set of concepts that complement ours and, as such, can be used as further enrich our theory. An overview of the identified concepts is documented in Table 4.3.

Note that concepts identified in the literature which emerge in our theory under a different category (*e.g.,* in [165] "parallel development" is categorized as a cause, rather than a consequence), are not considered as complementary concepts to our theory. In fact, such divergence is exclusively due to the perception of our participants and the applied coding strategy (see Section 4.2.2.2), rather than a concrete difference of content. The remaining of this section is dedicated to a further discussion of the literature review findings.

Table 4.3 – Complementary concepts to our theory identified in the literature

Source	Concept	Type	Category	Definition
Martini *et al.* [164]	**Suboptimal Reuse**	Implementation ATD	ATD item	Very similar code (if not identical) in different parts of the system, which is managed separately and not grouped into a reused component.
Martini *et al.* [164]	**Suboptimal APIs**	Implementation ATD	ATD item	Suboptimal design of APIs or the misuse of them, *e.g.,* methods that contain too many parameters, or components calling private APIs instead of public ones.
Martini *et al.* [164]	**Non-uniformity of Patterns**	Process ATD	ATD item	Patterns and policies that are not kept consistent throughout the system. *e.g.,* different name conventions applied, or different design / architectural patterns used to implement similar functionalities.
Martini *et al.* [164]	**Superfluous Testing**	Product Development	Consequence	Unnecessary tests performed due to ATD.
Martini *et al.* [165]	**Dependency Violation**	Implementation ATD	ATD item	The presence of architectural dependencies (for example at different component levels) which are considered forbidden in the (context-specific) architecture.
Martini *et al.* [165]	**Uncertainty of use cases in early stages**	External	Cause	Difficulty in defining a design and architecture that has to take in consideration a lot of unknown upcoming variability.
Martini *et al.* [165]	**Spit budget for project and maintenance**	Internal	Cause	Separation of available budget into two distinct budgets, one dedicated to development, and one to refactoring activities.
Martini *et al.* [165]	**Non-completed refactoring**	Internal	Cause	Non-completed refactoring activity, introducing new ATD.
Besker *et al.* [72]	**Incapability to Address Quality Requirements**	ATD item	Process ATD	Lack of mechanisms for addressing non-functional requirements, *i.e.,* lack of an implementation assuring quality requirements and the lack of mechanisms to test them.
Besker *et al.* [72]	**ATD Detection, Identification, Measurement, and Monitoring**	Active	Management Strategy	Tool-supported processes aimed at the identification and management of technical debt specific to the architecture of software-intensive systems.

Martini *et al.* [164] present a multi-case study adopting some GT techniques, while our investigation systematically applies the GT methodology. Accordingly, the two works use different techniques for data collection, incident coding, and results synthesis (cf. Section 4.2 of this study and Section 2 of [164]). Regarding the results, [164] presents a taxonomy of ATD items and a model of their effects: the specific *ATD items* reported in Table 4.3 are complementary to the ones emerging in our theory; the *effects* are categorized into *causes*, *phenomena*, and *extra activities* and the specific concepts resemble the categories *cause* and *ATD management strategy* emerging in our theory, which in turn resulted in a richer number of categories *e.g.,tool*.

A previous work of the same authors [165] zooms into the evolutionary nature of ATD and its accumulation and refactoring over time, *e.g.,* the causes specific to accumulation. Our work is complementary by emphasizing the theoretical structure underlying ATD instead. Overall, similarities and complementarities are promising for a future comparative analysis between the results of [164, 165] and our substantive theory, with the ultimate goal of formulating a formal theory. A formal theory is the widest form of GT, constructed by using formal concepts. Such theoretical construct applies to the conceptual area it has been developed for, and commonly spans over a set or family of several substantive areas [167]. In our case, a formal theory could potentially regard the role that architectural technical debt plays in the implementation and maintenance of software-intensive systems.

Besker *et al.* [72] conducted a systematic literature review to define a descriptive model of ATD. By comparing the findings of such study with our theory, we can observe a noticeable gap between the results of the two studies. In fact, numerous aspects reported in the model of [72], such as *ATD detection*, *ATD identification*, *ATD measurement*, *ATD monitoring* and related concepts, did not emerge in our theory. Rather than attributing the absence of such concepts to unsaturation, we conjecture that such divergence in results is due to the research methodology followed. In fact, we can observe that the missing concepts are related to ATD aspects which, while actively discussed in academic settings (e.g. *ATD identification* [1]), did not yet get traction in industry (*e.g.,* see [R-Q8]). From this finding we can conclude that more action research is needed to bridge the gap between studying ATD and dealing with it in practice.

A broader review of the literature shows that the most studied type of technical debt is *source-code ATD* [1] [168], such as ATD related to component dependency [118] or modularity [43]. This typology of ATD emerged in our theory as a specific concept of the *ATD Item* category, namely *implementation ATD*. This category is also mentioned

in Brooks's popular book "The Mythical Man Month" [169], where a recurrent theme is to "plan to throw one away", *i.e.,* designing a system (and organization) by envisioning change, as it will eventually happen. Moreover, the *workaround that stayed* ATD item is extensively discussed in Fowler's book titled "Refactoring: improving the design of existing code" [170], again with a primary focus on TD at the source code level. The "re-inventing the wheel" ATD item is instead discussed in Szyperski's book [171], where design reuse is advocated as the practice of sharing certain aspects of an approach across various projects, thus avoiding to re-invent the wheel across projects and organizations. The book also presents various techniques for addressing this ATD item, e.g. using software libraries for sharing solution fragments, interaction and subsystem architectures. Other kinds of ATD items, such as *segments of code affected by TD* have been studied exclusively in narrower pockets of research [1] [168] [45], and are mapped to our category *new context, old architecture*. In [172], Martini *et al.* identified the information required to prioritize ATD. By comparing their findings to our theory emerges again the current lack of awareness of research findings in industrial contexts, as in our theory prioritization emerged as a mere "gut feeling" (see Section 4.3.8). The literature further investigates other emerging categories, such as TD management strategies [87], and the impact of TD on morale [173], but does not systematically focus on the architectural level as we do.

4.5 Theory Evaluation Results

In this section, we document the evaluation results of our theory, carried out by leveraging the focus group method presented in Section 4.2.3. Specifically, we base the evaluation of our theory on the four criteria presented by Glaser [139], as we followed such GT stance to construct our theory. In the following, the assessment results of each evaluation criterion is discussed separately.

4.5.1 C1: Theory *Fit* to Underlying Data

This first criterion evaluates if the categories of the theory are a good representation of the underlying data, *i.e.,* if the categories are able to suitably characterize the incidents collected for this study. By inspecting the incidents collected via the grounded theory method, we observed that, while minor facets and details of the incidents were seldom missing in the theory documentation, all data points resulted to be represented in the theory. Additionally, via the focus group method, we observed that the theory is also well-suited to fit new data related to the elements of the the-

ory, as recurrently participants not only recognized all the theory elements, but also provided additional examples of them according to their personal experience.

4.5.2 C2: Theory *Workability*

This criterion assesses if the theory is able to *work*, *i.e.,* to explain and support reasoning on the phenomenon under study. In the focus group sessions, participants recognized from their experience the elements reported in the theory, and only in a few cases further clarifications were required to detail the meaning of a concept, which was afterwards acknowledged (*e.g.,* in the case of the "*TD Halo*" ATD Item). Recurrent sentences expressed by participants such as *"I recognize them [theory elements] a lot"*, and *"I have examples of this [theory element]"* provided us confidence that the theory provides a faithful representation of the phenomenon, is relatable by practitioners, and is able to work in practice. Strengthening the achievement of theory workability, during the focus group sessions, we noted that participants recurrently adopted elements of the theory, such as category, types, and relations, to frame their own examples, reason about their experiences, and discuss about potentially missing elements.

4.5.3 C3: Theory *Relevance*

The third criterion of Glaserian GT evaluation entails the assessment of the *relevance* to action a theory possesses in the area it purported to explain. In order to analyze this criterion, during the focus groups, a dedicated discussion was conducted on how the practitioners would use the theory in their current practice. According to participants, the theory eases the *communication* and sharing of knowledge related to ATD in practice, by providing a common terminology to use, and a methodical view of how the phenomenon is structured, which is often lacking in industrial contexts. This enables practitioner to adopt a shared lexicon of ATD, rather than adopting an individual one, and leveraging an encompassing overview on how such concepts are related, in order to collectively reason on ATD instances.

Secondly, practitioners detailed how the theory provides the ability of gaining *awareness* of ATD in practice, enabling them to understand in a systematic way the ATD they are facing, put it into perspective, and gain further insights into what is happening.

Another element pointing to the relevance of the theory is its use for *training*. Participants described that, while the notions present in the theory may be familiar to senior

software architects, these are not well known by junior colleagues. By utilizing the theory as the basis for training, it is possible to provide less experienced practitioners with knowledge on ATD, to gain further understanding on the phenomenon, and manage it collectively with deeper familiarity in present and future occurrences.

Finally, participants expressed interest in adopting the theory for *analysis and documentation* purposes, either to (i) assess the current state of ATD and analyze situations (*e.g.,* via a checklist representing the elements of the theory), (ii) include the theory in their of documentation practices, or (iii) detect ATD instances based on the symptoms documented in the theory.

4.5.4 C4: Theory *Modifiability*

The last criterion entails the evaluation of the *modifiability* of the theory as new data appears. In order to evaluate this criterion, we assessed if our theory on ATD was modifiable according to the new concepts that emerged during the focus group discussions. This led to the modification of the theory by including 12 new concepts discussed by the focus group participants (depicted with the icon in Figures 4.7-4.13), and additional insights in other already present concepts (*e.g.,*, the relation between external and internal causes). We note that, while new concepts were introduced in the theory, and other concepts were modified, the "kernel" of the theory, *i.e.,* its categories, types, and relations, remained unvaried. This further confirms the attainment of theoretical saturation in the GT study, while proving the modifiability of the theory as new data appears.

4.6 Verifiability and Threats to Validity

We ensure the anonymity of our participants, their companies, and their collaborators. Hence, we keep confidential their identifying details, under the human ethics guidelines governing this study.

Accordingly, and as customary in grounded theory (*e.g.,* [147]), the verifiability of our results should derive from the soundness of the research method followed. We therefore provide in Section 4.2 an in-depth description of the research method we followed throughout our investigation, and (within space constraints) reference as much as possible to direct quotes from our participants (albeit excerpted).

A potential threat to validity is the theoretical sensitivity of the principal investigator (cfr. Section 4.2.1). In fact, the author resulted to be already exposed to the ATD research body of knowledge for one year prior the study execution. Nevertheless, we do not deem this as a major threat to our investigation, as the relatively limited exposure provided the researcher sufficient knowledge to improve his sensitivity, while limiting the possibility to introduce preconceptions and concepts consolidated during multiple years of experience in the field. In order to mitigate this threat, all the authors of this study refrained from investigating the literature till after the establishment of our theory.

A threat to generalizability of our results is entailed by the sample of participants that took part in this study. As detailed below, the presented theory has not to be considered as absolute or final, as it emerged from the experiences and knowledge of the involved participants, with additional considerations extrapolated from the state-of-the-art academic literature. To mitigate this threat, we interviewed practitioners from 22 distinct companies of different sizes and working in different domains. By conducting focus groups, we assessed that this threat did not appear to significantly affect the version of the theory established before the focus groups were conducted. Hence, we remark that this threat may potentially affect with a higher probability the results of the focus group method.

As any grounded theory study, our investigation establishes a mid-range substantive theory, that is, a theory where elements belonging to the studied context can be transferred to other contexts with similar characteristics [136]. We hence do not claim our theory to be absolute or final, and we highly welcome its extension, *e.g.,* by adding detail to emerging concepts of our theory, or even unveil new concepts and categories that did not emerge in this investigation.

4.7 Conclusion

Our investigation provides empirical insights into the challenges faced by practitioners when dealing with ATD. From our study emerge eleven interrelated categories regarding ATD, leading to a cohesive theory of ATD that connects its causes, consequences, symptoms, management strategies, etc. We made a deep-dive into those categories by grounding our findings in the knowledge of experienced software practitioners. Notably, among other results, from our investigation emerge sets of symptoms, consequences, and management strategies on which future research, methodologies, and tooling, can be based. By carrying out an evaluation of the theory

via focus groups, we confirmed that the theory fits its underlying data, is able to work, has relevance, and is modifiable.

A research avenue we find particularly interesting exploring is the *further study of ATD symptoms*, with particular emphasis on quantifiable ones, in order to determine which symptoms are best suited as foundation for novel ATD identification and management techniques, *e.g.,* by leveraging the method presented in [54]. Another interesting research direction is about the definition of methods and techniques to (i) *automatically* identify the components of the system which require immediate attention from the ATD perspective (we call them *ATD hotspots*) and (ii) *recommend* developers which actions should be taken for paying off the ATD accumulated in those components. Additionally, we are interested in studying the use of the theory in practice, *e.g.,* by conducting case studies with industrial partners and ad-hoc assessments of ATD instances via our theory. Finally, as discussed in Section 4.4, we are interested in combining the theory built in this paper with other complementary theories in order to build a *unified formal theory* of architectural technical debt.

Acknowledgments

We express our sincere gratitude to the 27 participants who took part in this study, for their time, support, and passion, which made this investigation possible. Additionally, we would like to thank Eltjo Poort, for his support. This research was conducted under the approval of the University of British Columbia Research Ethics Board, application number: H19-01125.

Part II

Architectural Technical Debt in Android Applications

In the first part of the manuscript, we studied the ATD phenomenon without focusing explicitly on any specific development context, technology, or software ecosystem. Nevertheless, some derivative findings of the first three chapters of the thesis pave the way for a novel research direction. From the literature study presented in Chapter 2, we observed that no ATD identification approach has been proposed to target specifically the Android ecosystem, even if it is to date the most popular mobile ecosystem[10]. From the findings of Chapter 4 we observed that the characteristics of ATD can vary greatly according to the specific context considered. Hence, to detect ATD at at a refined level of granularity, it is possible to develop ATD identification and management techniques that focus on specific technologies, or software ecosystems. Focusing on a specific development context allows to consider ATD instances that do not appear in other contexts, and to examine ATD items specific to certain technologies and development frameworks. Finally, from the findings of Chapter 3, we observed that ATD analyses can be successfully shaped according to the specific context at hand.

Based on the findings of the first part of the thesis, we dedicate this second part to investigate TD and ATD specific to the Android ecosystem. To the best of our knowledge, TD and ATD in the Android ecosystems remains a very marginal research branch, considered exclusively in the work presented in this thesis. Hence, as no other supporting literature was available to us, we started to investigate the topic by conducting a preliminary study, presented in the following chapter (Chapter 5). In such chapter, opening our investigation on TD and ATD in the Android ecosystem, we assess the evolution of one of the most prominent quality attributes impacted by TD, namely maintainability. This provided us a better understanding of the context at hand, on which Android-specific ATD studies could be based. In Chapter 6 instead, we propose an ATD identification approach to tackle ATD in the Android ecosystem, and a set of architectural guidelines through which ATD can be identified, managed, and prevented.

[10] https://gs.statcounter.com/os-market-share/mobile/worldwide

5 How Maintainability Issues of Android Apps Evolve

"If I test out android," Phil Resch prattled, "you'll undergo renewed faith in the human race."

Philip K. Dick
"Do Androids Dream of Electric Sheep?"

This chapter is based on I. Malavolta, R. Verdecchia, M. Bruntink , B. Filipovic and P. Lago, *On the Evolution of Maintainability Issues of Android Applications*, IEEE International Conference on Software Maintenance and Evolution (ICSME), 2018. [51]

This chapter constitutes the “bootstrap" investigation which opens the research thread specific to the investigation of architectural technical debt of Android applications. Specifically, the research presented in this chapter reports the answer to the fourth research question of this thesis (RQ4). In order to answer RQ4, we designed and conducted an empirical study on 434 GitHub repositories containing open, real (i.e., published in the Google Play store), and actively maintained Android apps. We statically analyzed 9,945 weekly snapshots of all apps for identifying their maintainability issues over time. We also identified maintainability hotspots along the lifetime of Android apps according to how their density of maintainability issues evolves over time. More than 2,000 GitHub commits belonging to identified hotspots have been manually categorized to understand the context in which maintainability hotspots occur. Our results shed light on (i) how often various types of maintainability issues occur over the lifetime of Android apps, (ii) the evolution trends of the density of maintainability issues in Android apps, and (iii) an in-depth characterization of development activities related to maintainability hotspots.

Contents

5.1 Introduction

Mobile apps dominate our world today, showing no signs of slowing down market growth anytime soon [174]. As of March 2017, there are more than 2.8 million Android applications available, with more than one thousand apps being published *everyday* [174]. Android applications are not only being published in large numbers, they are also being consumed by users in large numbers. According to the official Android developer portal [175] there are more than 1.5 billion downloads from Google Play Store *every month.* A platform of such a large scale leads to an extremely crowded market and fierce competition. If developers are to succeed in such a competitive environment, it is of paramount importance that the mobile apps they produce are of extremely *high quality.*

Software maintenance is the activity of modifying a software product after its delivery in order to improve performance, add functionalities or perform corrective tasks on the existing product [176]. Software maintainability can be defined as the property of software that provides insights about how easily a software system (in this case an Android app) can be maintained [176]. In principle, apps with higher maintainability can be released and updated with less effort and provide the users with high quality features. Maintenance can be seen as one of the most important activities within the mobile app lifecycle. For example, updates of widely popular mobile apps like Facebook are consistently published *on a daily basis* [177]. Apart from the official Android guidelines [178], there is very little evidence about Android apps maintainability.

In this chapter we present a *large-scale empirical study on the evolution of statically-detectable maintainability issues across the Android ecosystem.* In particular, we refer to "maintainability issue" as code that is classified as high risk by the **Software Analysis Toolkit (SAT)** [179], a toolset that was developed by the Software Improvement Group (SIG) [180].

From a methodological perspective, we firstly built a dataset of 434 Android apps that are (i) open (i.e., available as open-source projects in GitHub), (ii) real (i.e., distributed through the Google Play Store), and (iii) actively maintained (i.e., no single-commit or toy projects). Then, we (i) mined their GitHub repositories and extracted 9,945 weekly snapshots[1] from 106,689 commits, (ii) analysed each snapshot via an industrial tool for static analysis, and (iii) identified the occurrences of 5 types of maintainability issues in each snapshot. Afterwards, we manually analyzed 1,230 apps for building

[1] A "snapshot" is defined as the state of a repository after a week of active development, i.e. a week in which at least one commit occurred.

a reusable taxonomy of the trends in which the density of maintainability issues of Android apps can evolve over time. In order to do so, we considered the notion of commit-level *issue density (cd)* [181] reported in Formula 5.1.

Moreover, we identified 3,374 maintainability hotspots, defined as the points along the lifetime of an Android app where developers are injecting an anomalous number of maintainability issues with respect to the current app size. Finally, in order to understand the context in which maintainability hotspots occur, we investigated on the (self-reported) activities performed by developers in the context of all identified maintainability hotspots. We carried out the latter step by manually inspecting and categorizing 2,112 GitHub commits belonging to the identified maintainability hotspots by conducting independent content analysis sessions.

The main **contributions** of this chapter are:

• a characterization of the frequency of maintainability issues in Android apps;

• a taxonomy and characterization of the evolution trends of the maintainability issues' density in Android apps;

• a characterization of maintainability hotspots for Android apps, together with an investigation about the Android development activities performed when those hotspots occur;

• the replication package of our empirical study containing its results, raw data, and mining- and data analysis scripts.

The **target audience** of this chapter includes both Android developers and researchers. We support developers by providing a set of actionable and evidence-based insights, which can be used for improving the maintainability of Android apps; we support researchers by objectively characterizing the state of the practice about maintainability of Android apps.

Chapter structure. Section 5.2 provides background information. Section 5.3 presents the design of the study. Section 5.4 presents and discusses obtained results. Sections 5.6 and 5.7 describe threats to validity and related work, respectively. Section 5.8 closes the paper and discusses future work.

5.2 Background

Software maintainability. This study relies on the software quality model defined in the ISO/IEC 25010 standard [182]. The standard defines a generic software quality model composed of the following characteristics: functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability and portability. There, **software maintainability** is defined as the degree of effectiveness and efficiency with which a product can be modified to improve it, correct it or adapt it to changes in environment, and in requirements [182]. The ISO/IEC 25010 software quality model further divides software maintainability into 5 sub-characteristics: modularity (degree of change impact of one component w.r.t. others), reusability (degree to which an asset can be used to build other systems), analyzability (extent to which a software product can be analyzed, with the goal of identifying parts to be modified), modifiability (extent to which a software product can be modified without lowering its quality), and testability (extent to which a software product can be tested).

Source code quality tools. In order to minimize maintenance costs, developers can track and improve their source code quality with the help of open-source tools. For example, **Lint** [183] is an Android Studio source code scanning tool. The Lint tool provides support in finding potential bugs and optimization improvements with respect to security, performance, usability, correctness and internationalization. Another open-source solution is **FindBugs** [184]. It reports nearly 300 different bug patterns in Java and reported bugs are categorized and assigned a priority level. **SonarQube** [185] is also a prominent open source static code analyzer. It focuses on the detection of bugs, code smells and security vulnerabilities. SonarQube currently supports more than 20 programming languages including Java and offers a vast range of customization parameters for ad-hoc analyses. **PMD** [186] is a popular source code analyzer, able to find common programming issues such as empty catch blocks, unused variables etc. PMD supports a variety of different languages, among which are Java and XML, making it suitable for analyzing source code of Android applications. Furthermore, PMD provides support for identification of duplicated code in Java source files. Another freely-available tool is **CheckStyle** [187]. Although it does not identify bugs, it allows Java developers to write code that adheres to coding standards, thus increasing code readability. **JArchitect** [188] is instead a commercial static analysis tool that, in addition to code metrics, provides capabilities to inspect the quality of software architecture through dependency analyses and validation of architectural rules. **SciTools Understand** [189] is another commercial tool aimed at providing an overview of a software product through a mix of code metrics analyses

and dependency visualization techniques.

In this study we use **SAT**, a toolset developer by SIG, a software consultancy company providing insights into software systems' source code quality. SAT is based on the SIG maintainability model, which defines metrics to measure ISO 25010 maintainability based on source code. SIG has used this model in a consulting practice for the past 10 years, measuring hundreds of systems and billions of lines of code. The model works by comparing individual code-level metrics to a database consisting of several hundreds of software systems, thus producing (relative) ratings, and finally combining the code-level ratings into ratings for the ISO 25010 maintainability characteristics (i.e., analyzability, modifiability, testability, modularity, reusability) [190].

Empirical studies have shown that the metrics used by the SAT tool are correlated with the maintainability issue resolution performance of software developers [191]. Since the model relies on source code measurements, rather than functional or behavioural characteristics of a software product [192]. As such, this model is a very good candidate for this study due to its compliance with the well-acknowledged ISO/IEC 25010 standard and to the full automation enabled by the SAT analysis tool. SAT allows us to automatically perform static code analysis on snapshots of multiple apps, and provides maintainability metrics for further statistical analysis.

5.3 Study Design

To provide objective and replicable findings, a complete replication package is available to reviewers as part of this submission[2], including the source code of all the developed mining and analysis software, and raw data.

5.3.1 Goal and Research Questions

The **goal** of this study is to *analyze* the source code of Android mobile apps *for the purpose of* characterizing their evolution and determining best practices *with respect to* its maintainability *from the viewpoint of* software developers, *in the context of* open-source apps published in the Google Play Store. The related research questions are described in the following.

RQ4.1 – *Which are the most recurrent types of maintainability issues in Android apps?*

[2]https://github.com/ICSME/ReplicationPackage2018

By answering this research question we can comprehensively characterize which are the most recurrent maintainability issues during the evolution of Android apps. This enables developers and researchers to get a better understanding of Android-specific maintainability issues through empirical evidence, lying the groundwork for the efficient management of maintainability issues in Android apps.

RQ4.2 – *How does the density of Android maintainability issues evolve over time?*

RQ2 investigates whether the evolution of the density of maintainability issues exhibits identifiable characteristics (i.e., specific trends). Specifically, it provides insights about *how* each type of maintainability issues tend to remain/grow/decrease in Android apps over time, potentially with a negative or positive impact on the overall maintainability of the app in future releases. The trends emerging from this study can guide developers in classifying their own apps, comparing them with others, and take action depending on the level of maintainability they want to achieve.

RQ4.3 – *What are the development activities in which maintainability hotspots occur?*

RQ4.3 aims at identifying the relation between the activities performed by developers and the occurrence of maintainability hotspots. Intuitively, a maintainability hotpot is an anomalously high value appearing in the maintainability time series of an app (refer to Section 5.3.4.3 for the formal definition of *hotspot*). The identification of those relations will help in understanding what are the Android development activities that are more sensible to the injection of each type of maintainability issues. Android developers can use this information for (i) better planning code refactoring sessions, (ii) better planning their code review sessions (e.g., steering the assignment of code reviews), (iii) taking special care of their code quality when performing tasks belonging to activities highly correlated with maintainability issues.

5.3.2 Context and Dataset

5.3.2.1 Context selection

Since this study is focused on *real-world Android apps for which we can track their maintainability and development activities over time*, the **context** of this study consists of a set of Android apps that (i) are freely distributed in the Google Play store and (ii) have their versioning history hosted on GitHub.

5.3.2.2 Dataset building

The dataset building process of this study is similar to the one proposed in [193]. As shown in Figure 5.1, we consider the following initial sources: GitHub, FDroid, and Wikipedia. From Github, a custom search was performed that targeted all the repositories containing a link to a Google Play Store app page in their readme files[3]. The second source for our dataset is FDroid [194], a largely known online catalogue of free and open-source Android projects. From this catalogue, a search was applied that locates apps that contain: a) a link to the respective GitHub repository, and b) a link to the respective Google Play store page. The third source is a Wikipedia [195] containing a maintained list of free and open-source Android applications. We performed a manual selection from this list. The merging step (1) of the three considered data sources resulted in a total of 9,400 apps.

Some of the apps were not published on the Google Play store, and were therefore excluded (2). This occurs if developers decide to remove the app from the store or if Google takes down apps for violating some publishing policies. Next, duplicate entries have been removed (3).

In the next filtering step we identified repositories containing actual Android app source code (4). This filtering step has been done by considering only the repositories containing the mandatory `AndroidManifest.xml` file.

Then, we filtered out repositories which did not contain an Android manifest file in the source code folder (5). The rationale for this step is that the folder containing the Android manifest file should also contain the complete source code for each application.

In order to avoid inactive or unmaintained repositories [100], we considered only repositories with at least 6 commits and having a lifetime span of at least 8 weeks (6). The 6-commits threshold corresponds to the median of the number of commits for all considered repositories before this filtering step, while the 8-weeks threshold comes from the fact that 8 weeks is the average development time for an Android app [196].

We further cleaned up the dataset by filtering out all those GitHub repositories con-

[3] In order to do not occur into the GitHub limit of 1,000 results per search, we stratified our search queries by date range so that each search resulted in less than 1,000 results. The whole set of considered dates ranges from the creation of GitHub (i.e., Jan 1, 2001) to the day in which searches were performed (i.e., May 2, 2017)

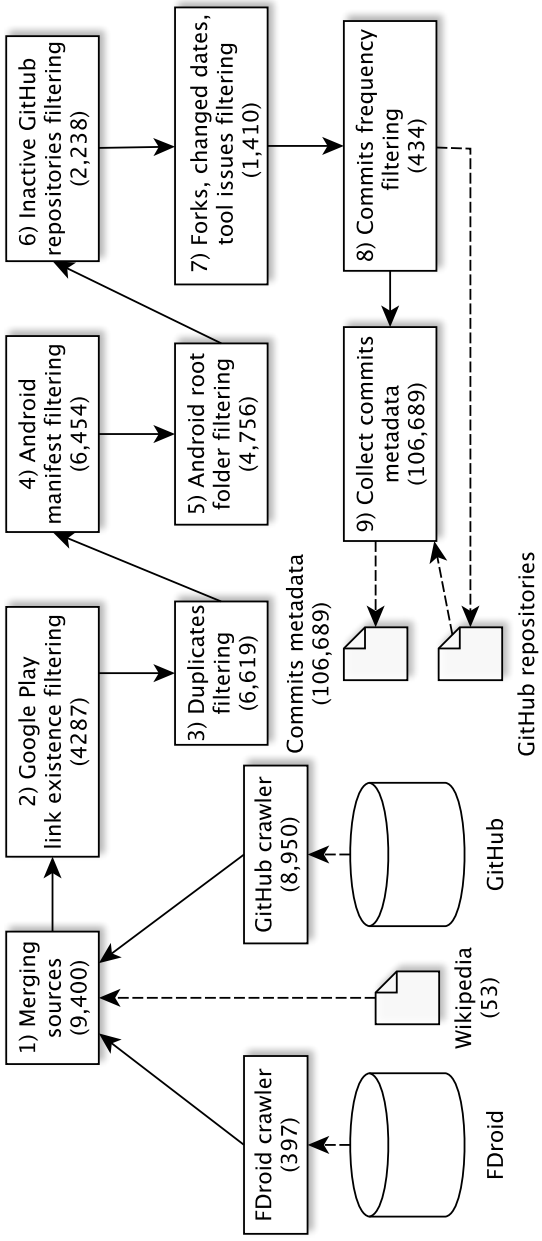


Figure 5.1 – The dataset building process

taining (i) commit dates which were manually modified by developers (our study has a strong emphasis on the time dimension), (ii) forks of other repositories (to avoid duplicates), and (iii) source code not analyzable by the SAT tool, e.g., Kotlin-based apps (7).

The last filtering step involved (i) the identification and removal of all the snapshots for which there were no commits in the GitHub repositories (in our study we exclusively focus on the active snapshots and we avoid the noise produced by periods of inactivity, e.g., due to holiday breaks) and (ii) the subsequent filtering of all the GitHub repositories having less than 8 snapshots after the snapshots removal (8). After this step, our final dataset contains **434** GitHub repositories containing open, published, and actively maintained Android apps, for which an analyzable commit history is available.

Finally, we extracted all commits of the main branch of each GitHub repository, leading to a total of **106,689** commits (9).

5.3.2.3 Demographics

In the following we provide an overview of the apps included in our dataset.

As shown in Figure 5.2, the median app of the dataset results to be developed for 16 **active weeks of development** (hereafter, snapshots).

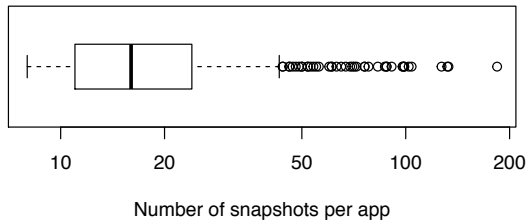


Figure 5.2 – Distribution of snapshots per app

As we may expect, the **number of commits per app** is characterized by a high variance, ranging from a minimum of 12 to a maximum of 2,407 commits per app. The median app of the dataset has 123 commits (see Figure 5.3).
The **number of commits per snapshot** varies from a minimum of 1 commit (which

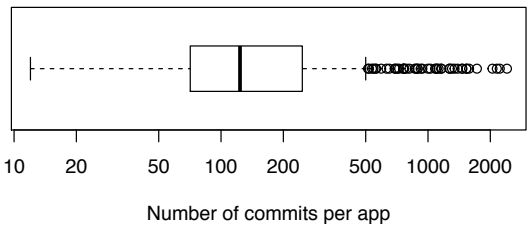


Figure 5.3 – Distribution of commits per app

was required in order to characterize a snapshot as active) to a maximum of 251 commits. The median snapshot is composed of 6 commits (see Figure 5.4).

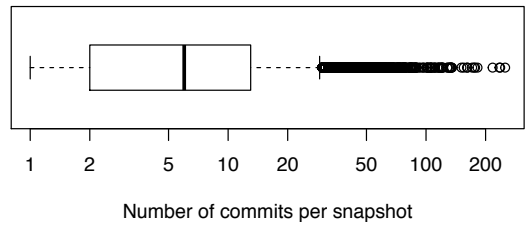


Figure 5.4 – Distribution of commits per snapshot

Regarding the **period of development**, we ensured that the apps within our dataset are heterogeneous, in order to do not bias our study due to some specific versions of the Android platform. In particular, the earlier app development start date is close to the initial release of the Google Play market (end of 2008) till early 2017 (close to when our crawling process was executed, see also Section 5.3.2). In Figure 5.5 the development start date of all the apps is reported.
Finally, by considering the **number of unique contributors** per app, the median shows that apps developed by 3 unique contributors result to be the most common in our dataset. Out of the 434 apps considered, only 72 resulted to be developed by a single contributor, making us reasonably confident that our dataset does not contain a large number of toy or demo apps (which usually are committed by a single developer). In Figure 5.6 the distribution of number of unique contributors per app is depicted.

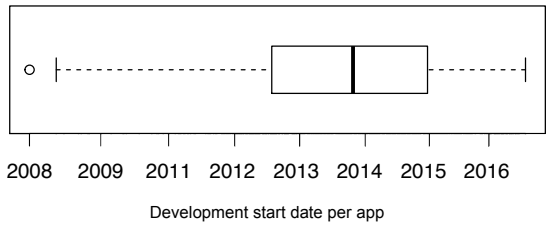


Figure 5.5 – Distribution of development start date per app

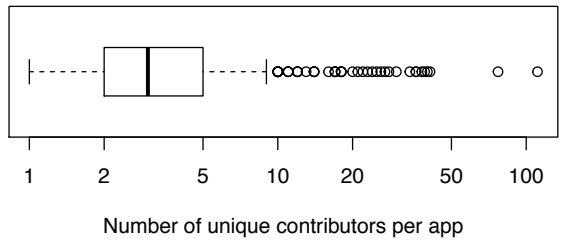


Figure 5.6 – Distribution of unique contributors per app

5.3.3 Data Extraction

Snapshots Extraction. This study considers the evolution of maintainability issues of an Android app as a sequence of snapshots $(S_1, S_2, ..., S_n)$, where a snapshot is defined as a set of source code files of an app at a given point in time. A time-windowing approach has been adopted and closely follows the approach presented in the work by Di Penta et al. [181]. A snapshot series can be extracted from an app's GitHub repository by considering all the commits performed between the begin and end of a snapshot interval. For this research, the time interval between two snapshots in a snapshot series is defined as one week, mainly because it has been empirically shown that Android apps are updated once a week or less frequently [197]. Once the snapshot series has been obtained, each app repository has been cloned and subsequently checked out for each snapshot in the series.

In total, the whole analysis process of extracting snapshots and applying static code analysis to each snapshot spanned a period of 2 months. Furthermore, a total of 800 million LOC has been processed, and more than 7 thousand GB of file system

resources has been considered across 98,487 snapshots.

Maintainability Issue Density Identification. In this step we used the Software Analysis Toolkit (SAT) to process every snapshot of each app, producing the source code metrics related to maintainability issues of Android apps. The total static code analysis processing of all snapshots took 12.45 days, with an average of 8.73 seconds per snapshot. As mentioned before, the metrics in SAT are based on the ISO/IEC 25010 quality model. The model underlying SAT measures the properties of a software product at four levels of abstraction, namely: unit, module, component and overall system level. In the context of this study, **unit** is a Java method, a **module** is a Java class, a **component** is a Java package, and a **system** is the whole app.

In this study, we consider the maintainability issues defined by SAT ([179]), namely:

- **Unit Size (US):** Units exceeding 30 lines of code[4].
- **Unit Complexity (UC):** Units exceeding 10 McCabe Cyclomatic Complexity [107].
- **Unit Interfacing (UI):** Units having more than 4 (formal) parameters.
- **Module Coupling (MC):** Modules exceeding 20 incoming dependencies (eg. method invocations, class extensions, interface implementations).
- **Duplication (DP):** Code clones of at least 6 lines of code. SAT detects type-1 clones [201] after a cleaning process that removes empty lines and comments.
- **Maintainability (MT):** The total count of occurrences of the previous issues types.

Once the maintainability issues have been identified, a unified and comparable measure of their amount is needed, in order to allow for objective comparisons between apps with different size and between different issue categories. Therefore, in this study we consider a notion of commit-level *issue density (cd)* [181], defined as follows:

$$cd^{i}_{c_a} = |\bigcup_{x \in I^{i}_{c_a}} x| / NKLOC_{c_a} \tag{5.1}$$

where i uniquely identifies one of the categories of maintainability issues according to the ISO/IEC 25010 quality model (e.g., unit size, unit complexity), a is the app being considered, $c_a \in C_a$ is a commit in the GitHub repository of a, $I^{i}_{c_a}$ is the set of identified issues of type i in the repository of a after checking it out at commit c, and

[4]The thresholds for maintainability issues have been defined based on the results of empirical studies involving the SAT tool [198, 199, 200].

$NKLOC_{a,c}$ is the number of thousands of lines of Java source code in the repository of a after checking it out at commit c. Intuitively, cd indicates the number of issues belonging to a specific category that are introduced by a commit, normalized by considering the current size of the repository. Being S_a the set of all weekly snapshots of a, the issue density $d^i_{s_a}$ of each $s_a \in S_a$ is defined as $cd^i_{c_a}$, where c is the last commit performed within the time window of s_a.

5.3.4 Data Analysis

5.3.4.1 RQ4.1

In order to test whether the issue types exhibit a significant difference w.r.t. issue density, we adopt the omnibus Kruskal-Wallis test, i.e., a non-parametric test for testing the difference among multiple medians. In order to avoid potential threats to validity due to fishing rate we adjust the significance level by means of Bonferroni correction [202]. In addition to the Kruskal-Wallis omnibus test, in order to assess if there is a significant difference between the pair-wise comparison of issue types, we conduct a series of two-tailed Mann-Whitney tests. This is not required in order to test the *null* hypothesis, i.e. that all means of the issue density among issue types is equal. Nevertheless we use this statistical tests to get further insights in the data. As the previous test, the Bonferroni correction is used to adjust the significance level. This latter process consists in dividing the significance level by the number of tests performed (i.e., the total number of possible pairwise-combinations of issue types).

Apps characterized by long-lasting active development might affect the results due to their high number of snapshots present in the dataset. In order to mitigate this potential threat to internal validity, the above presented tests are carried out both on the complete dataset of snapshots and the median values of the snapshots aggregated per app. Due to space limitations, and as the results do not significantly differ, in the result section we report the results relative to the analysis of the complete dataset.

5.3.4.2 RQ4.2

For each app a and maintainability issue i, we firstly create a time series representation TS^i_a by temporally ordering all the density values d^i_s across all snapshots $s \in S_a$ of a. The time of the first and last observations of each TS^i_a are set to the timestamp of the first commit among the set of all commits C_a of a and the timestamp of the end of the time window identified by the last snapshot in S_a, respectively. The period

of each TS_a^i has been elicited by (i) building the periodogram of TS_a^i (defined as the vector of frequencies at which the spectral density of a time series is estimated [203]), (ii) considering its dominant frequency f, and (iii) converting f to the time domain by dividing f by 1.

As initial exploration, we check if the obtained time series exhibit a stationary behaviour. From a statistical perspective, the mean and variance of a stationary time series are constant over time [204] (i.e., it has no trend over time). We apply to each TS_a^i the Augmented Dickey-Fuller test (ADF) [205] (with $\alpha = 0.05$), which is a unit root test for stationarity with H_0 = *the time series has a unit root (it is non-stationary)* and H_1 = *the time series does not have a unit root (i.e., it is stationary)*. We adjust the obtained p-values via the Bonferroni correction since we are applying the ADF test 6 times, one for each type of maintainability issue. From this preliminary test, apps result to be mostly non-stationarity for all types of maintainability issues, motivating us to further inspect their exhibited trends.

We decompose each TS_a^i into its seasonal, trend and irregular components [206] using the STL algorithm [207]. Intuitively, given a time series, STL iteratively extracts its seasonal component by lossless smoothing of the seasonal sub-series (e.g., the series of the first values of all seasons, of the second values of all seasons, etc.). Then, the seasonal values are removed, and the trend component is extracted by smoothing the remainder. The irregular component is computed as the residuals from the seasonal plus trend fit [207]. We use the STL algorithm as it does not assume any distribution of the time series, it has been successfully used in previous software engineering studies [202, 208], and an efficient implementation is available as an open-source R package[5].

For answering RQ4.2, we perform a qualitative study on the plots of the trend components of all TS_a^i. Since the manual analysis of all the collected trend components is infeasible, we randomly selected a sample composed of 205 apps and analyzed their trend component for each type of maintainability issues[6] (summing up to 1,230 distinct trends), and manually scrutinize and categorize them into relevant groups by applying the open card sorting technique [209]. To minimize bias, two researchers have been involved in this activity and the results have been checked by a third researcher. This activity resulted in a two-levels taxonomy of maintainability issues trends. In Section 5.4.2 we present each category in the taxonomy and its frequency

[5] http://stat.ethz.ch/R-manual/R-devel/library/stats/html/stl.html

[6] By considering 205 apps we achieve a 95% statistically significant sample of the 434 apps of our dataset with a 0.05 confidence interval.

within our dataset.

5.3.4.3 RQ4.3

We answered RQ4.3 by following two main phases. The first phase targets the *identification of maintainability hotspots* along the lifetime of an Android app. Given an app a and its density time series TS_a^i (one for each type of maintainability issue), the set of maintainability hotspots H_a^i is defined as follows:

$$H_a^i = \{s_j \mid (d_{s_j}^i - d_{s_{(j-1)}}^i) > \sigma(TS_a^i), \qquad j = 2, \ldots, |TS_a^i|, s_j \in TS_a^i\} \tag{5.2}$$

where $d_{s_j}^i$ is the density of the maintainability issue of type i in snapshot s_j and σ is the standard deviation of the density values along the time series TS_a^i. In other words, we consider as hotspot every snapshot s_j in which the density of a maintainability issue i w.r.t. its preceding snapshot $s_{(j-1)}$ is higher than the standard deviation of the whole time series of i. We build upon this standard-deviation-based strategy since (i) it is not feasible to build upon more advanced models for outliers detection (e.g., ARIMA [210]) for all 434 apps since they require delicate manual tuning for each app and (ii) it is computationally efficient. Despite its apparent simplicity, an independent manual exploration of a subset of the apps by two researchers confirms that this strategy proves effective in correctly identifying maintainability hotspots. This phase led to the identification of 3,374 maintainability hotspots over 46,873 GitHub commits.

In the second phase, we consider GitHub commit messages as indicators of the actual activities performed by developers and we *manually analyze the commits performed during the occurrence of maintainability hotspots.* As manually analyzing all 46,873 GitHub commits is unfeasible, we build a representative sample of commits for each type of maintainability issue (95% confidence level, 0.05 confidence interval), leading to a set of 2,112 unique commits to be manually analyzed. To this aim, we conduct content analysis sessions [211] on all 2,112 commit messages and categorize them according to the taxonomy of self-reported activities of Android developers proposed and empirically validated by Pascarella et al. [212].

For answering RQ3, we present and discuss (i) the frequency and distribution of maintainability hotspots within the whole dataset of Android apps and (ii) how frequently each category of Android developers' activities appears in maintainability hotspots (i.e., it was assigned to a commit message during the content analysis), leading to the identification of those development activities which are potentially

more related to maintainability issues. Finally, we reuse the publicly-available dataset of 5,000 manually categorized commits produced in [212] as ground truth for all commits i.e., commits either belonging to maintainability hotspots or not. We test the possible relationship between the activities performed in all commits and those performed in commits belonging to maintainability hotspots; we use a Chi-Square test to assess the relationship and Cramer's V to establish the effect size [202].

5.4 Results

In this section we report the results which emerged from the analysis of the gathered data.

5.4.1 RQ4.1. *Which are the most recurrent types of maintainability issues in Android apps?*

To answer *RQ1* we inspect how the maintainability issues of the collected apps are distributed among the different maintainability issues considered. We therefore plot in Figure 5.7 the violins plots of the complete distribution of issues per type.

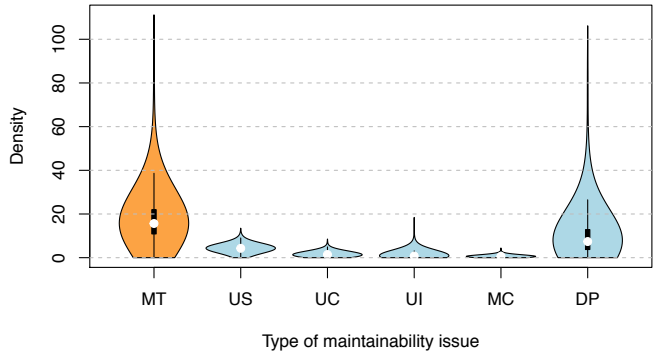


Figure 5.7 – Distribution of unique contributors per app

By considering the totality of maintainability issues, we see that on average almost 18 issues, spread out throughout the different issue types, are present every NKLOC

(M=15.65, m=17.87)[7]. The aggregated maintainability issue distribution is reported in the leftmost violin plot of Figure 5.7. From the remaining violin plots we observe that the issue density varies among the different issue types. From the Kruskal-Wallis omnibus test we evince that the distributions significantly differ (p-*value*$< 2.2 \cdot 10^{-16}$). We hence reject the *null* hypothesis, i.e. that all means of the issue density among issue types is equal. From an additional pair-wise comparison between issue types we see that all issue types occur with different rates. The most recurring maintainability issue results to be *duplication* (M=7.393, m=10.230), followed by *unit size* (M=4.290, m=4.249), *unit complexity* (M=1.4510, m=1.6150), *unit interfacing* (M=0.7736, m=1.1030), and *module coupling* (M=0.6258, m=0.6746). Apart from *duplication* and to a certain extent *unit size* violations, the remaining issue types present only small differences in distribution.

> *Code duplication* is the most recurrent maintainability issue in Android apps, followed by *unit size violations*, *unit complexity violations*, *unit interfacing violations* and *module coupling violations*. In our dataset, the overall maintainability of the analyzed apps is highly impacted by *code duplication* issues.

Not all code duplicates are bad [213]. We conjecture that the higher frequency of *code duplication* issues is primarily due to the Android programming idiom and code duplication in Android can be related to the templating phenomenon due to the activity-intent-based idiom of the Android programming model. Nevertheless, idiom-based templating can lead to introducing bugs (if not carefully used) and overlooking inconsistencies [213], which might be remarkably detrimental for the maintainability of mobile apps.

5.4.2 RQ4.2. *How does the density of Android maintainability issues evolve over time?*

Firstly, we assess if the density of maintainability issues exhibits a stationary trend in time. We applied the ADF test for obtaining the number of stationary and non-stationary apps for each type of maintainability issue. As shown in Table 5.1, the ADF test reveals that 93.01% of all apps exhibit a non-stationary behaviour (p-*value*

[7] Where M is the median value and m the mean value.

< 0.008), whereas only 6.99% are stationary in at least one type of maintainability issue.

Table 5.1 – Number of stationary and non-stationary trends per maintainability issue type

Maintainability issue	Stationary	Non-stationary
MT	**1**	**433**
US	4	430
UI	61	373
MC	67	367
UC	37	397
DP	12	422
TOTAL	182 (6.99%)	2,422 (93.01%)

This finding provides evidence that, according to our dataset, *the density of maintainability issues in Android apps is not stable over time.* This means that Android developers actually have an impact on the overall maintainability of their apps over time. Under this perspective, the instantaneous value of $d^i_{s_a}$ can be used for keeping the maintainability of apps under control and taking informed decisions when planning for maintainability-related activities (e.g., refactoring sessions).

The vast majority of apps do not exhibit a stationary behavior across all types of maintainability issues.

As the majority of apps does not exhibit stationarity, we further inspect their trends as detailed in Section 5.3.4.2. The manual analysis of the 1,230 trend components led to the elicitation of the taxonomy presented in Table 5.2. The taxonomy objectively represents the categories in which the density of maintainability issues of Android apps can evolve. The taxonomy is composed of two levels, where the first one represents trends with similar overall characteristics (e.g., density growth or reduction), whereas the second level represents trends at a finer grain (e.g., stable increase, valley). Figure 5.8 shows an example of each trend category of the taxonomy.

Table 5.2 – Taxonomy of evolution trends

Trend category	Description
Growth (G)	In the first weeks the app has a low density of maintainability issues, followed by an overall non-decreasing trend
Increase (I)	The density follows a generally increasing trend (some minimal decreasing parts could be present)
Stable Increase (SI)	The density is relatively low in the initial weeks, then it gradually increases throughout the whole lifetime of the app
Increasing Plateau (IP)	The density reaches plateau(s) after an overall non-decreasing trend
Plateau Increasing (PI)	The density is stable and relatively low in the first weeks, then it increases for the whole lifetime of the app
Reduction (R)	In the first weeks the app has a high density of maintainability issues, followed by an overall non-increasing trend
Decrease (D)	The dual of I
Stable Decrease SD)	The dual of SI
Decreasing Plateau (DP)	The dual of IP
Plateau Decreasing (PD)	The dual of PI
Mixed (M)	Mixed trend where the density of maintainability issues grows and declines over time
Hill (H)	The density is relatively low in the initial weeks, it gradually increases up to a certain peak, and then it gradually decreases
Valley (V)	The dual of H
Anomalous (A)	It does not fall into any of the previously defined categories
Constant (C)	The density of maintainability issues is the same over time
Constant (C)	Same as C

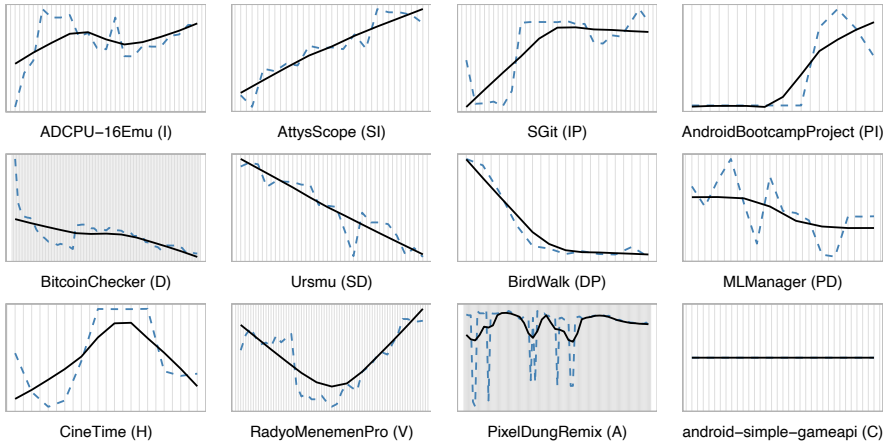


Figure 5.8 – Examples of evolution trends of the density of maintainability issues of Android apps

In order to effectively interpret the manually gathered results, we represent the frequency of each trend across issue types in the heatmap in Figure 5.9. We evince that the trend category G (composed of I, SI, IP, and PI) is the most recurrent. For the majority of maintainability categories new issues are introduced over time in the apps by following different increasing trends. Only seldom are maintainability issues resolved.

Of the subcategories composing G, IP is the most recurring one. We conjecture that this is the result of the introduction of maintainability issues in the early stages of development, followed by a "saturation period". This later period of the IP trend is attributed to a higher awareness that is given by developers to maintainability issues after reaching a certain level of technical debt. From this recurrent trend we observe that developers, while not actively resolving issues, avoid to introduce more in the more mature stages of the app, potentially before maintainability issues become unmanageable in number. Issues characterized by an initial period of stability before increasing (PI) are less frequent: if maintainability issues are introduced, usually it starts from the early development stages.

The only exceptions opposing the higher occurrence of the G trend can be observed for the issues of type UI and DP, which exhibit more frequently an H trend. This can

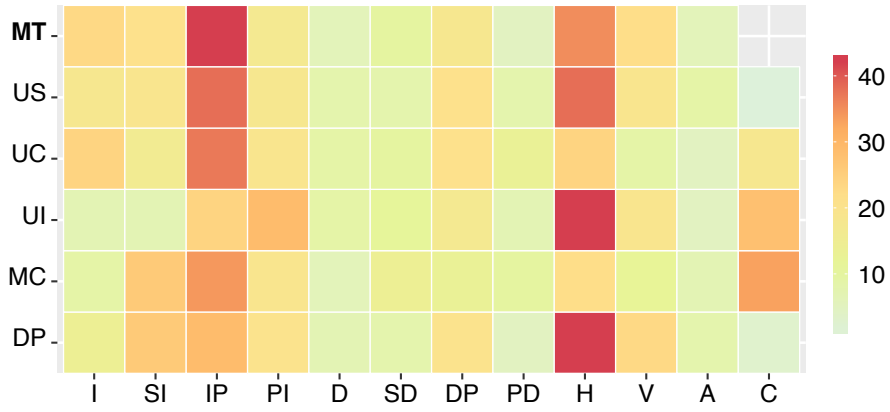


Figure 5.9 – Evolution trends by maintainability issue types

be caused by the nature of such type of issues. In fact, UI issues manifest themselves as Java methods become more complex, directly impact development time required to use them, and can be easily spotted by developers. Hence, after a period of initial growth, effort might be spent to resolve such issues and avoid cumbersome development activities in the future. A similar reasoning can be applied to the H trend of DP issues: while due to time constrains DP issues can be acceptable at initial stages of development, in time resolving such apparent issues can be a valuable activity which eases future development.

From the lower frequency of R trends (D, SD, DP, PD) we evince that only seldom development activities lead to a consistent decrease of maintainability issues. This may be attributed to scarce effort put into structured refactoring processes aimed to improve and maintain apps' software quality.

Constant trends also result be less occurrent. We can conjecture that this is due to the selection criteria adopted to systematically filter the weeks of development considered in our dataset. In fact we selected exclusively active weeks of development, i.e. when changes were carried out on the apps, which reflected in a changing number of issues.

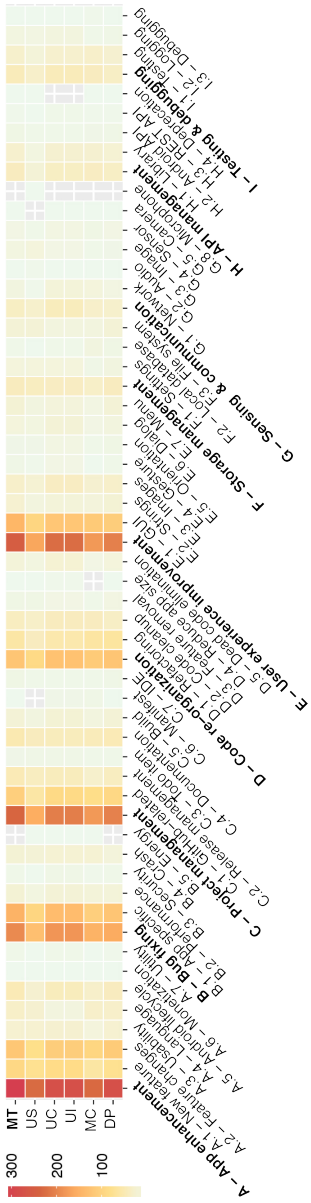


Figure 5.10 – Frequencies of Android development activities performed during a maintainability hotspot

The scarce occurrences of anomalous trends (A) shows that our taxonomy proved to be effective, and encompassed the vast majority of the trends of maintainability issues analyzed.

> *Unit interfacing* and *code duplication* issues result to be outliers, displaying a *hill* like trend, which can be motivated by the nature of these issues. Only seldom maintainability of apps exhibits an overall *decrease* or *constant* trends in time.

5.4.3 RQ4.3. *What are the development activities in which maintainability hotspots occur?*

As anticipated in Section 5.3.4.3, we identified a total of 3,374 maintainability hotspots involving 46,873 GitHub commits. Their descriptive statistics are reported in Table 5.3.

Table 5.3 – Descriptive statistics for the hotspots of each maintainability issue type per app

Issue type	Min.	Max.	Median	Mean	SD	CV
MT	0	6	1.0	1.38	1.03	0.75
US	0	6	1.0	1.35	1.03	0.76
UC	0	8	1.0	1.16	1.09	0.94
UI	0	8	1.0	1.16	1.09	0.94
MC	0	7	1.0	1.31	1.16	0.88
DP	0	6	1.0	1.41	1.12	0.79

SD = standard deviation, CV = coefficient of variation.

Overall, the number of maintainability hotspots per app ranges between 0 and 8, with a median of 1 (similar values for the mean and standard deviation). The low values for standard deviations also tell us that the distribution of the number of hotspots across apps is very compact, with a strong tendency towards one hotspot per app.

Maintainability hotspots do not occur often in the lifetime of each app. Nevertheless, on average each app has at least one maintainability hotspot.

Now we zoom into the activities performed by developers during the occurrence of maintainability hotspots. To do so, we manually categorized 2,112 commit messages according to a taxonomy of self-reported Android development activities[212]. The taxonomy entails a wide variety of different activities at different levels of abstraction (e.g., bug fixes, functionality implementation, release management, access to sensors, etc.). The taxonomy is composed of two levels, where the first layer groups together activities with similar overall purpose (e.g., app enhancement, bug fixing, API management), whereas the subcategories (49 items) in the lower level provide a finer-grained categorization [212]. Figure 5.10 shows the frequency of each category of Android developers' activities across the various types of maintainability hotspots[8]. The colour of each tile reports the sum of all occurrences of each development activity in correspondence of a hotspot of a specific type of maintainability issue (note that multiple activities can be assigned to each hotspot). The order of development activities follows the rank of most frequent activities in Android apps, as emerged in [212] (e.g., group A is more frequent than group B, activity A.1 is more frequent than activity A.2).

Overall, our categorization follows the same trends identified in [212], with the most prominent categories of activities being *app enhancement* (specially for new and updated features), followed by (app specific) *bug fixing*, and *project management*. Interestingly, the category *user experience improvement* is quite frequent in the presence of hotspots (+7.28% more than it is for "standard" commits, i.e., commits unrelated to hotspots), specially for those activities related to the graphical user interface (E.1 - GUI). This result can be an indication that Android developers should pay special attention when working on the business logic related to buttons, UI layouts, event listeners, etc., as those activities are potentially more related to the presence of maintainability hotspots. Moreover, we noticed an increase also for the category *Android lifecycle* (+2.86% w.r.t. all commits), which refers to the activities about the management of Android components lifecycle events and transitions (e.g., the `onCreate` of `Activity`). Finally, also the *Documentation* category exhibits an increase in frequency when maintainability hotspots occur (+ 3.23%); this kind of activities refers

[8] In Figure 5.10 we depict in bold the top level categories and in plain text all the subcategories of the taxonomy.

to adding/refining comments in the source code and working on the documentation of the app (e.g., description of the app, its requirements, UI mockups). This result happened for 85 commits and was quite unexpected as in principle this category should not be related to maintainability at all. A deeper investigation reveals that all snapshots containing the 85 documentation-related commits include also other commits, which may be related to other development activities playing a role in the occurrence of hotspots. Nevertheless, the co-occurrence of documentation-related activities and maintainability hotspots may be an indication that commits chronologically close to documentation-oriented activities are correlated with hotspots. We leave this analysis for future work.

In order to better characterize how Android development activities may be correlated with maintainability hotspots, we test if there is a significant difference between the frequencies in our categorization of 2,112 commits and the ones observed in the 5,000 manually categorized commits in [212]. It is important to note that the commits categorized in [212] have been randomly selected, they can belong either to maintainability hotspots or not. For all maintainability issue types we obtained a statistically significant measure of correlation between these two categorizations ($p-value < \sigma$, where $\sigma = 0.008$ because of the Bonferroni correction), allowing us to reject the null hypothesis that the two categorizations are independent. The Cramer's V test reveals a small effect size for all maintainability issue types ($0.19 \leq V \geq 0.22$). Together, those results (i) confirm our preliminary exploration that the frequencies of Android development activities in commits belonging to maintainability hotspots are in line with those involving all commits and (ii) such a relationship is only weak.

> Maintainability hotspots in Android apps tend to occur independently of the type of development activities performed by developers. Activities related to the *GUI* and the management of the *Android lifecycle* are slightly more prone to co-occur with maintainability hotspots.

5.5 Discussion

5.5.1 Observations

The obtained clusters constitute a foundation for having an up-to-date overview of maintainability trends of Android apps, bringing a number of interesting observa-

tions. Firstly:

> Most of the applications follow a rather stable evolution trend, with higher oscillations in the initial phases of the project.

Indeed, the most noticeable differences and instability in evolution trends were found in the first quarter of the development timeline of an app, suggesting that this period is the most unstable and brings the most source code changes. This is understandable, as app development evolves rapidly at the beginning of the project, with numerous functionalities being added or removed throughout development; it is also understandable that few changes to the source code appears at later stages of the timeline when the app has been likely already published or almost finished.

When looking at the identified clusters of evolution trends of each category of maintainability issues:

> Unit-related categories (i.e., unit size, complexity, and interfacing) follow similar evolution trends, with slight changes in the magnitude of their respective issue densities.

However, within the three unit-related issue categories, *unit size* and *unit complexity* exhibit similar trends both in evolution and in density. *Unit interfacing* exhibits more stability with respect to the evolution trends, containing numerous apps with stable low densities over time. The density of the *duplicated code* issue is generally much higher than other categories of issues. This finding suggests that:

> Duplicated code poses numerous threats to maintainability of Android apps, as it constitutes most of the overall maintainability issues. Duplicated code blocks in general are not trivially removed, implying that these redundant code blocks can cause significant maintenance efforts at later development stages.

The duplication evolution trends suggest that duplicated code that is introduced at the beginning of the application development becomes increasingly harder to

identify and remove over time, as the decreasing trend noticed in the highest density cluster exhibits only a slight decrease. Furthermore, with the noticeable constant growth, whether sharp or steady, in the remaining clusters, it can be said that:

Regarding *code smells*, the stable cluster in this category contains most of the apps and shows the lowest density values of the three identified evolution trends, suggesting that:

> Developers seem to already address code smells effectively.

In the cluster exhibiting a sharp decline evolution trend, it can be seen that apps that start out with a high density of *code smells* swiftly decrease the density over time. The same decrease can be noticed once again after the small increase in the same cluster, suggesting that developers may have more ease in removing these code smells issues from their apps. Planned bug fixes are common in the software development, and could also explain the decreases in density of *code smells* issues over time.

We found also several interesting implications regarding maintainability issues across Android components. For what concerns Android activities, we can observe that

> Android **activities** heavily suffer from code *duplication* (the highest proportion across the whole study).

If on the one hand this finding can imply that Android developers should be proactive in avoiding to duplicate code when developing the activities or their apps in order to, e.g., avoid having to change many parts of the source code when fixing a bug in an activity. On the other hand, this finding may be unveiling an intrinsic problem of the Android programming model, which is leading developers to duplicate code within activities. A better investigation of this result is left for future work.

We noted that Android services exhibit the highest number of maintainability issues per component, suggesting that:

Android **services** can contain a high number of *code smells* (they have the highest proportion in the whole study, together with broadcast receivers) and *code duplication* is recurrent as well.

This finding can be due to the fact that services do not have a user interface and developers may have the perception that a bug in an Android service may not be perceived by the user. We are planning to investigate on this reflection via a qualitative study involving developers interviews.

Considering that Android broadcast receivers are generally used as a means of communication between different apps, we suggest to keep these components as simple as possible. Indeed our study reveals that:

Android **broadcast receivers** exhibit a low number of maintainability issues per component, however, they exhibit the highest percentage of *code smells* of the whole study and a fairly high percentage of *unit size* and *duplication*.

The implication above is related to the so-called *broadcast receiver's connector envy*, presented by Bagheri et al. in [214]. Indeed, Bagheri et al. empirically discovered that Android broadcast receivers *poorly separate concerns, resulting in deficiencies that affect maintainability and efficiency*; this finding is also confirmed by our study, where the most recurrent maintainability issues in broadcast receivers are related to code smells, duplication, unit size and unit complexity.

Content providers receivers exhibit the least average amount of maintainability issues per component, however we can notice surprising trends with respect to the other Android components. Specifically:

Android **content providers** exhibit the highest percentages of issues related to *unit size*, *unit interfacing*, and *unit complexity* of the whole study.

We can conjecture that the number of issues related to *unit interfacing* is higher because content providers heavily rely on the use of interfaces to allow communication data sharing between apps. Content providers also exhibit the highest amount of *unit size* and *unit complexity* issues compared to other components. This finding

may suggest that more attention should be paid to managing the size and number of interfaces of Android content providers with respect to other Android components.

5.5.2 Best Practices for Android Developers

The following best practices can serve as a guidance for new and experienced Android developers. This set of best practices is *actionable*, meaning that it can be considered as a pragmatic checklist for reducing the overall application maintenance efforts, alongside improving overall app's source code quality. The Android maintainability best practices extracted from this study are the following:

- BP_1 : Developers should avoid duplicating code in their apps, as it is one of the most recurrent and severe maintainability issues, which may cause significant maintenance efforts at later development stages.
- BP_2 : Developers should pay special attention to duplicated code when working on Android activities, as they are especially keen to suffer from this kind of issues.
- BP_3 : Developers should pay special attention to code duplication while developing Android services, alongside code smells (for the same reason as BP_2).
- BP_4 : developers should pay special attention to checking the quality of the code of broadcast receivers, possibly using analysis tools such as Android Lint or FindBugs.
- BP_5 : Developers should keep content providers manageable with respect to their size, complexity and the amount of input parameters (for the same reason as BP_1).
- BP_6 : Developers should avoid introducing complex logic in Android content providers and keep their source code as simple as possible from the beginning of the project.
- BP_7 : Developers should measure and track the codebase of the app early from the beginning of the project with respect to its maintainability in order to keep predictable maintenance efforts.

5.6 Threats to Validity

Construct validity. The results of this study are based on the current implementation of SAT, the used static code analysis tool. It is hence paramount that the tool was implemented and configured correctly. This major threat was mitigated through different strategies. Firstly, the tool documentation is made available in order to

detail the maintainability issue detection processes and the configuration settings adopted. Furthermore, interviews have been conducted with the tool developers to investigate the details related to the identification of the identified maintainability issues. Additionally, an inspection of several detected issues across issue categories was performed manually. Finally, the SAT tool is utilized on a daily basis in industrial settings, and was also utilized in previous researches carried out by independent researchers [215], [216], [217].

An additional threat to construct validity is constituted by the representativeness of the selected apps. This threat was mitigated by carrying out an in-depth data quality assurance process (reported in Section 5.3.2.2). In addition, we ensured that the data was encompassing and heterogeneous in terms of development lifespan, number of commits, and number of contributors etc. (see reported in Section 5.3.2.3).

Conclusion validity. The most prominent threat to conclusion validity is constituted by the data extraction and analysis processes adopted to gather the results. In order to mitigate such major threat to conclusion validity, we strictly adhered to a set of *a priori* defined data extraction and analysis processes. Such processes were explicitly conceived to gather and analyze significant data to answer our research questions. The totality of the data extraction and statistical data analysis processes are reported in their entirety in Section 5.3.3 and Section 5.3.4 respectively. In addition, a replication package with the raw data and statistical data analysis scripts is made available for the complete reproducibility of the results.

The majority of the statistical tests produced sound p-values, far below the chosen significance level of 0.05. To minimize the error rate of the results, the Bonferroni correction was adopted to adjust the significance level, when required. In order to further mitigate potential threats to conclusion validity, the data analysis process was jointly discussed by the researchers and the results were inspected independently. The level of agreement, especially for the manual labeling processes, was assessed by means of the Cohen's kappa statistics. Disagreements were jointly discussed in order to scrupulously align the data extraction and analysis processes and ensure a high quality level of the gathered results.

Internal validity. As repositories containing the app source code differ in structure, it is possible to obtain false results with the inclusion of non-app related source code (e.g., third party libraries or code developed for other platforms). This threat has been mitigated both before and during the static code analysis. Firstly, the root app folder containing the Android source code has been identified for each app repository, and subsequent repository metrics have been collected only for the source code

contained within this directory. Also, the SAT tool allows the selection of different files during static code analysis, and this was exploited in the sense that .jar files and library directories have been excluded from the analysis.

Another threat relates to the exclusion of component-related maintainability metrics from the measurements. In order to ensure the envisioned quality of the analyzed data, an in-depth data quality assurance process was carried out (see Section 5.3.2.2). All instances presenting inconsistencies led to the total exclusion of the entire app data from which the instance belonged to. In this way, we were able to strictly control the quality of the analyzed data by including exclusively the apps of which every piece of data adhered to our quality criteria.

External validity. In this study we consider a set of Android apps sampled from a real-world setting. This was possible by considering exclusively Android apps which are published on the Google Play Store. In order to further mitigate potential threats to external validity, we ensured that the apps considered were representative of the apps present in the Android ecosystem. From an inspection of the gathered dataset the apps resulted to be highly heterogeneous in terms of size, development lifetime, number of contributors etc. (see section 5.3.2.3). We hence deem our dataset as representative of the current trends identifiable in current Android mobile apps.

This research considers Android apps published in GitHub since we require access to their full versioning history. We do not target app binaries in the Google Play store as it only provides the latest release of each app. Further, we are interested in the maintainability issues introduced by developers in the Java code of their apps; in Google Play only the binary code of the app is available, which may be structurally different from the source code produced by developers (e.g., because of code obfuscation). Nevertheless, due to the high heterogeneity of the dataset (see Section 5.3.2.3) and the presence of all the considered apps in the Google Play Store, we do not deem this as a major threat to external validity.

5.7 Related Work

The state of the art on the maintainability evolution of Android apps is quite scarce, yet it exhibits some related work on Android source code quality and software evolution.

Hecht et al. [197] presented an approach for performing static code analysis on

Android app's bytecode and detecting software antipatterns. They analyzed the evolution of the quality across 3,568 versions of 106 different Android apps obtained from the Google Play Store. They identified relationships between antipatterns and five different quality evolution trends. This work ties into our research with a similar methodology and focus on quality aspects and their evolution in the context of mobile (Android) apps. Differently, we focus on maintainability-related issues (rather than software antipatterns) and on how development activities are related to them. Our research scope involves the analysis of over 400 Android apps, compared to the 106 analyzed in [197].

Di Penta et al. [181] analyzed the evolution trends of statically detectable vulnerabilities of software projects. For the detection of vulnerable source code lines, they have used 3 different static code analysis tools, namely Splint, Rats and Pixy. Three different networking systems were analyzed by means of executing the static code analysis tools on different snapshots of the system. This study is methodologically similar to ours. However, the subjects and therefore the outcomes of their study differ from ours, as we are specifically focussing on Android apps and their maintainability, as opposed to vulnerabilities in source code of generic software systems.

Tufano et al. [218] investigated on *when* and *why* code smells are introduced in a software project. Their study involves investigating the circumstances and rationales behind bad code smells introduction, and is conducted on a change history of 200 open-source projects. Our research is specific to Android apps, and thus our results are more fine-grained with respect to the ones obtained in [218]. The focus of our study is on maintainability-related issues at a higher level of abstraction (i.e., units, models, and components) w.r.t. Tufano et al. who focus on fine-grained code smells at the level of source code.

Similar to our research, Koch [219] set out to analyze the evolution of open-source software systems on a large scale. Utilizing the data of 8,621 projects coming from SourceForge, the evolutionary behaviour of the systems was characterized by applying both linear and quadratic models to the systems, where the quadratic model outperformed the linear one. Furthermore, the evolutionary behaviour has been modelled as a function of lines of code and time since the first commit. Both Koch's and our study focus on large-scale, open-source systems. However, we focus on Android apps and Android-specific development activities.

5.8 Conclusion and Future Work

In this opening chapter of Android specific ATD, we uncovered the frequency and evolution of the overall maintainability issues of Android apps. Our results show that *code duplication* is the most recurrent maintainability issue (RQ4.1), which is intrinsic in the Android programming model and can be mitigated by a more careful programming style. In general, notwithstanding the issue type, maintainability issue density grows until it stabilizes, but issues are seldom fully resolved, which represents an important hidden lack of quality. Also, maintainability hotspots are independent from the type of development activity (RQ4.3), which means: *whatever you do, your development style will matter*.

In the closing chapter of this thesis, we further narrow our research scope by presenting an approach to automatically detect ATD present in Android apps, and a set of architectural guidelines aimed at guiding the identification and mitigation of Android-specific ATD.

6 Identifying Architectural Technical Debt in Android Applications through Compliance Checking

The first 90 percent of the code accounts for the first 90 percent of the development time. The remaining 10 percent of the code accounts for the other 90 percent of the development time.

Tom Cargill

This chapter is based on:
R. Verdecchia, *Identifying Architectural Technical Debt in Android Applications through Compliance Checking*, IEEE/ACM International Conference on Mobile Software Engineering and Systems (MobileSoft), 2018 [52]
R. Verdecchia, I. Malavolta, and P. Lago, *Guidelines for Architecting Android Apps: A mixed-method Empirical Study*, IEEE International Conference on Software Architecture (ICSA), 2019, [53]

Chapter 6. Identifying Architectural Technical Debt in Android Applications through Compliance Checking

This chapter reports the investigation carried out to answer our last research question included in the thesis (RQ5). Specifically, this chapter presents an approach aimed at automatically identifying Architectural Technical Debt in Android applications. The proposed approach leverages different techniques, ranging from architectural guidelines extraction, to the establishment of a reference architecture via metamodeling, architecture reverse engineering, and compliance checking. The content of this chapter documents an ongoing research endeavor, and includes (i) the outline of an ATD identification approach tailored for the Android context, and (ii) a mixed-method empirical study combining semi-structured interviews and multivocal literature review to systematically synthetize a set of 42 evidence-based Android architectural guidelines.

Contents

6.1 Introduction

In the past decade a drastic media consumption shift towards mobile devices took place. As a result, smartphones and tablets are nowadays more used than desktop computers [174]. It is hence not surprising that the development of mobile applications experienced an exponential growth in recent times. The shift towards mobile development was supported by the advent of dedicated *app stores*, such as Google Play and Apple App Store, where millions of mobile applications are available nowadays. Mobile application development results to be a highly competitive business, which on one hand can lead to high profits, but on the other hand it is sensible to the introduction of errors which may have a tremendous financial impact [220]. The mobile application business model is tightly coupled with users satisfaction, who can efficiently express their opinions through app reviews and rating systems. It is hence paramount, in order to ensure the user satisfaction and revenue of mobile applications, to be able to promptly and efficiently release new versions of mobile apps in order to introduce new features, fix bugs, and in general rapidly adapt to users' needs.

By considering the fast pace at which mobile applications need to evolve, Architectural Technical Debt (ATD) results to be a crucial yet implicit factor of success. This led recently to the official release of a set of Android architectural components aimed to lower the apps complexity and provide a recommended Android architecture[1].

ATD is defined as sub-optimal design decisions which hinder the evolvability and maintainability of software applications over time. By identifying, resolving, and monitoring ATD of mobile applications, it is possible to enable them to rapidly adapt according to users' needs. While ATD has a substantial impact on the overall quality of software systems, its presence is hard to uncover due to its complexity and lack of tool support [8].

In this research we present a novel approach, based on architecture reverse engineering and compliance checking, for identifying ATD hotspots in Android applications. The presented approach is, to the best of our knowledge, the first one conceived specifically to identify ATD in Android applications.

[1] https://developer.android.com/topic/libraries/architecture

6.2 Approach Overview

In the literature several techniques aimed to identify ATD can be found [50]. Among these, a typology of approaches focuses on identifying ATD by comparing the architecture of the implemented software applications with a reference architecture. Occurrences where the implemented architecture is non-compliant to the envisioned reference architecture are regarded as potential ATD Items (ATDIs). This typology of approaches is particularly interesting as design guidelines can be directly embedded in the reference architecture. Building on such concept, the reference architecture can even be composed exclusively of architectural guidelines aimed at avoiding ATD. The approach presented in this research, conceived to identify ATDI of Android applications, is based on this intuition. The approach consists of five steps, namely: (1) architectural guidelines extraction, (2) establishment of an Android reference architecture, (3) reverse engineering of implemented architecture, (4) compliance checking, and (5) quantitative assessment of compliance violations. In the remainder of this section the steps constituting the approach, depicted in Figure 6.1, are further detailed.

6.2.1 Step 1: Android architecture guideline extraction

The first step of the approach consists in the identification of architectural guidelines that Android applications should adhere to in order not to incur in potential ATD. The data sources adopted for the extraction of architectural guidelines to construct the Android reference architecture are complementary and heterogeneous, in order to be as encompassing as possible, and consist of:

• **Official Android Guidelines:** Official Android documentation available online, such as the *Guide to App Architecture*[2];

• **Academic researches:** Peer-reviewed research papers considering architectural guidelines of Android applications, e.g. the study of Bagheri et al. [221];

• **Grey literature:** Non-academic writings on the topic available online, e.g. articles featured in Android related websites and blogs;

• **Developer interviews:** Semi-structured interviews with Android developers to validate and complement the data extracted from the above mentioned data sources.

[2] https://developer.android.com/topic/libraries/architecture/guide.html; Accessed 27 February 2018.

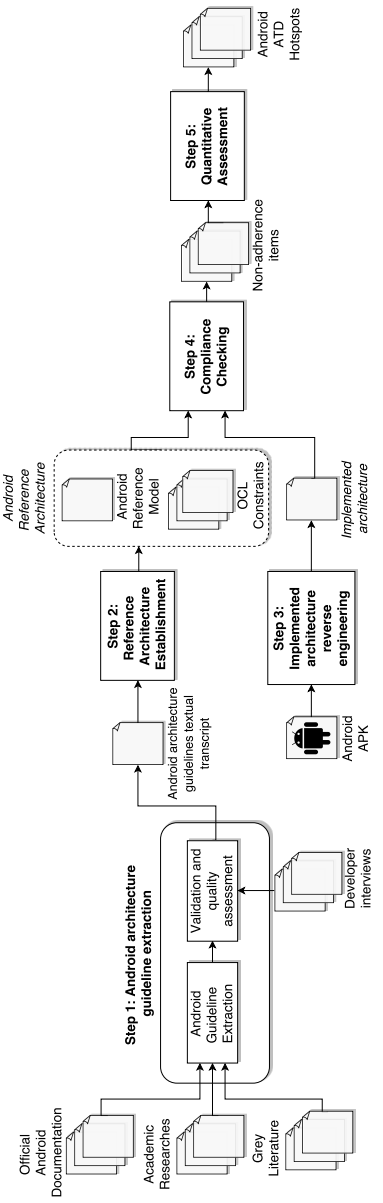


Figure 6.1 – Android ATD hotspot identification approach overview

A quality assessment is conducted on the extracted architectural guidelines in order to ensure the soundness of the findings, remove possible duplicates, etc.

The output of this step is a textual transcript of the identified Android architectural guidelines.

6.2.2 Step 2: Android reference architecture establishment

This step consists of the formalization of the textual transcript produced in Step 1. This process is required in order to effectively format the data for the subsequent automated analysis described in Step 4. Specifically, this step consists in developing a software model which conveys the information of the architectural guidelines extracted in Step 1. The resulting model is referred to as *Android reference model*. The reference model conforms to a chosen architecture description language (ADL). In particular, the Acme ADL[3] results suited for this process.

In addition to the Android reference model, in order to complement it with the information which cannot be exhaustively represented in form of a software model, a set of constraints expressed through the Object Constraint Language (OCL) are defined. The combination of the Android reference model and OCL constraints is what is jointly referred to in this document as *Android reference architecture*.

The output of this step is an Android reference architecture composed of Android architectural guidelines in form of a software model and complementary OCL constraints.

6.2.3 Step 3: Reverse engineering of implemented architecture

This step consists in the retrieval of the architecture of an implemented Android application through the analysis of its source code or `APK`. Specifically, this step is constituted by the automated reverse engineering of the most prominent Android architectural components (i.e. *Activities*, *Services*, *Content providers*, and *Broadcast receivers*) of an Android application and the relations between such building blocks in terms of connectors and ports. This process was first proposed by Bagheri et al. [221], who also provided empirical evidence of its effectiveness. Due to the potential complexity of this process, this step has to be carried out by utilizing dedicated

[3] http://www.cs.cmu.edu/ acme/

architecture reverse engineering tools, e.g. the *ACME-Generator* tool[4] available online. Note that to ease the compliance checking process (Step 4) both the ANdroid reference architecture and the implemented architecture must either (i) adhere to the same metamodel or (ii) be linked by a suitable model-to-model transformation being able to bridge models conforming to them.

The output of this step is the reverse engineered architecture of an implemented Android application.

6.2.4 Step 4: Compliance checking

Subsequent to the establishment of the Android reference architecture and the implemented architecture, a compliance checking process is carried out. During this process, items of non-adherence of the implemented architecture w.r.t. the Android reference architecture are identified and stored for subsequent analysis. Due to its complexity, this step has to be carried out through the aid of tools, e.g. the model comparison tool EMFCompare[5]. In order to carry out a sound compliance checking process and reduce the number of potential false positives, the compliance checking has to consider a semantic comparison logic. This can be achieved by utilizing the extension and customization mechanisms offered by the majority of modern model comparison tools.

The output of this step is the set of the non-adherence items of the implemented architecture w.r.t. the Android reference architecture.

6.2.5 Step 5: Quantitative assessment of compliance violations

Once the set of non-adherence items is computed, it is possible to analyze the gathered data to identify which architectural elements of the implemented architecture violate the highest number of Android architectural guidelines. Such identified items, referred to as *Android ATD hotspots*, are stored for a final manual inspection to prioritize them, select which require refactoring, etc.

The output of this step is the set of Android ATD hotspots, i.e. the components of the implemented architecture which contain the highest number of non-adherence items w.r.t. the Android reference architecture.

[4] https://github.com/arsadeghi/ACME-Generator. Accessed 27th February 2018.
[5] https://www.eclipse.org/emf/compare/. Accessed 27th February 2018.

6.3 Guidelines for architecting Android apps

Android is accounting for more than 85.9% of global smartphone sales worldwide [222], leading thousands of developers to choose Android as their first go-to development platform [223]. In the last quarter of 2018 more than 2.6 million Android apps were available in the Google Play, the official Android app store [224].

For surviving in such a highly competitive market, it is fundamental for app developers to deliver apps yielding high quality in terms of e.g., performance, energy consumption, user experience. Developers are investing great efforts to deliver apps of high quality and with short release times. In this context, a well-architected Android app is beneficial for developers in terms of maintainability, evolvability, bug fixing (e.g., resource leaks), testability, performance, etc. The most recent releases of the Android platform are putting more and more emphasis on the architecture of the apps, with a special focus on architecturally-relevant components[6], such as those belonging to Android Jetpack[7], the recently introduced collection of Android software components. However, *how to properly architect Android apps is still highly debated and subject to conflicting opinions*, usually influenced by technological hypes rather than objective evidence.

The goal of this research is twofold: (i) to characterize the state of the practice on architecting Android apps and (ii) to provide a set of evidence-based guidelines for supporting developers while architecting Android apps. Given the relatively low maturity of the subject and its tight connection with industry, we apply a *mixed-method empirical research design* that combines (i) semi-structured interviews with Android practitioners in the field, and (ii) a systematic analysis of both the grey (e.g., websites, on-line blogs, etc.) and white literature (i.e., academic studies) on the architecture of Android apps. Specifically, starting from 5 interview transcripts and an initial set of 306 potentially-relevant primary studies, through a rigorously-defined and replicable process, we select 44 data points, i.e., either interview transcripts or primary studies belonging to the grey/white literature. We analyze each data point in order to characterize how developers architect Android apps, what architectural patterns and practices Android apps are based on, and their potential impact on quality attributes such as maintainability. Finally, a set of 42 guidelines for architecting Android apps is systematically synthesized from the obtained practices. The emerging guidelines are organized around 4 themes including the most adopted architectural patterns and

[6] https://developer.android.com/topic/libraries/architecture
[7] https://developer.android.com/jetpack

principles when developing Android apps (e.g., Model-View-ViewModel[8]). The main **contributions** of this study are:

- interviews of 5 practitioners that provide qualitative information about architecting Android apps;
- a systematic analysis of the grey and white literature about architecting Android apps;
- a set of 42 evidence-based guidelines for architecting Android apps;
- the replication package of the study.

The **target audience** of this chapter includes both Android developers and researchers. Specifically, this study benefits (i) developers by providing evidence-based guidelines for taking action towards improving the architecture of their Android apps, and (ii) researchers by objectively characterizing the state of the art and practice about architecting Android apps.

The remainder of the chapter is organized as follows. The design of this study is presented in Section 6.3.1, followed by the reporting and discussion of the main results in Section 6.3.4. Threats to validity and related work are described in Sections 6.3.5 and 6.3.6, respectively. Section 6.3.7 closes the chapter.

6.3.1 Study Design

In this section we report the research questions (Section 6.3.2) and the steps of our mixed-method study (Section 6.3.3).

6.3.2 Research questions

***RQ*5.1:** *Which are the **general characteristics of the architecture of Android apps**?* This question can be refined into:

***RQ*5.1.1**: *Which **architectural patterns** are considered for architecting Android apps?* This research question aims at understanding which architectural styles/patterns[9] are considered for architecting Android apps. This provides us with a better understanding of current Android architecture practices, and sets the context for the next

[8] https://developer.android.com/topic/libraries/architecture/viewmodel

[9] For the sake of space, from this point onwards, "architectural style" and "architectural pattern" will be jointly referred to as "architectural pattern".

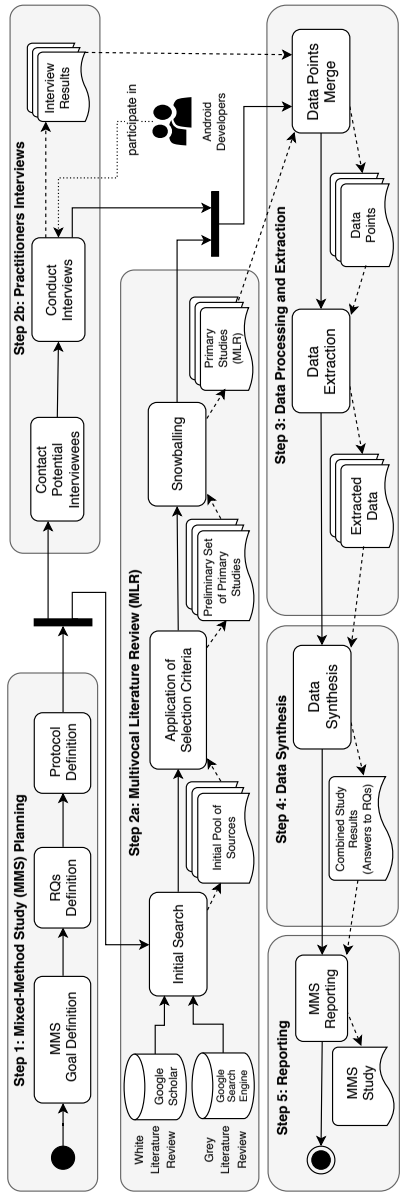


Figure 6.2 – Mixed-Method Study (MMS): process overview

research questions.

***RQ*5.1.2:** *Which **libraries** are referenced while considering the architecture of Android apps?* With this question we aim at identifying the programming libraries regarded as most influential while architecting Android apps. This provides further data on the technologies considered in order to support Android architecting processes.

***RQ*5.2:** *How to **guide developers** when architecting Android apps?* This research question constitutes the core of the study. By answering it, we aim at synthesizing a set of architectural guidelines for architecting Android apps. This provides practitioners with actionable guidance when architecting their Android apps, and supports researchers in future investigations on Android architecture.

***RQ*5.3:** *Which **quality requirements** are considered when developing and reasoning about the architecture of Android apps?* Today developing apps of high quality is fundamental for surviving in the Android market. This research question aims to understand which quality requirements (QRs [225], e.g., performance, usability, maintainability) are taken into account when dealing with the architecture of Android apps. This provides a good understanding of which QRs are potentially impacted the most by architectural decisions.

6.3.3 Research Method

In order to answer the research questions, we adopt a *mixed-method* approach consisting of a *multivocal literature review* [226] integrated with the results of semi-structured interviews with Android practitioners. An overview of the entire process is shown in Figure 6.2, while Figure 6.3 shows the number of selected data points in each step. The remainder of this section describes the key individual steps, Step 2a through 4.

6.3.3.1 Multivocal Literature Review (MLR)

The literature review is performed by rigorously following well established guidelines for conducting software engineering literature reviews [74, 227]. The guidelines are complemented by additional ones specifically targeted for the inclusion of grey literature in multivocal studies [228]. To have full control over the number and quality of the literature considered, the literature review is designed as a multi-stage process, reported below.

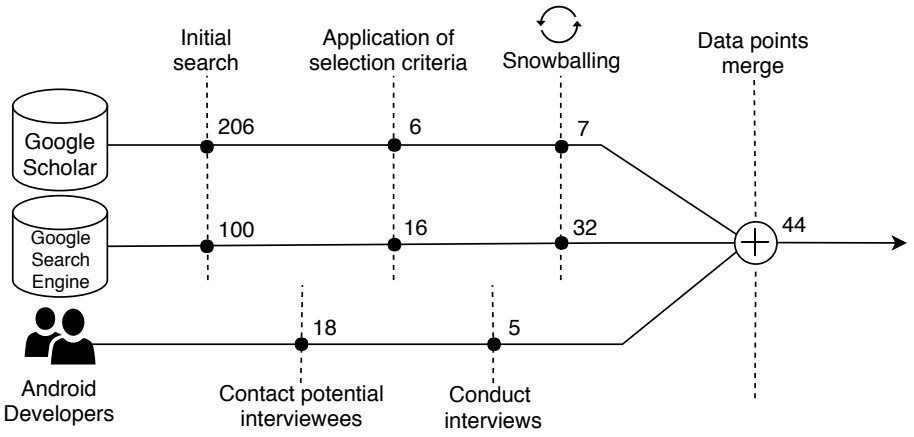


Figure 6.3 – Steps of the Mixed-Method Study (MMS)

Initial Search. In order to identify the potentially relevant white literature (WL) studies, a research query is executed on Google Scholar. We opt for such digital library as (i) its adoption constitutes a sound choice to determine the initial set of literature for snowballing processes [229], (ii) from a preliminary execution of the search query it results to be more inclusive w.r.t. Scopus and IEEE Explore, and (iii) the results of the query can be processed automatically via tool-support.

Listing 6.1 shows the search string we use. The query is purposely designed to be generic, in order to be as encompassing as possible while selecting a significant set of potentially relevant studies. The execution of the query for the WL returns 206 hits. Regarding the initial search of grey literature (GL), the query reported in Listing 6.1 is executed on the regular Google Search Engine by omitting Google Scholar specific syntax (i.e., the "intitle" keywords). The search engine is selected in accordance to the recommendations for including GL in software engineering multivocal reviews [228]. Due to the high volume of returned results, we limit the search to the top 100 results as stopping rule. This number proves also to be the theoretical saturation point [228] of the returned results.

```
(intitle:architecture (*@\textbf{OR}@*) intitle:
  architectural (*@\textbf{OR}@*) intitle:architect (*@\
  textbf{OR}@*) intitle:architecting) (*@\textbf{AND}@*) (
  intitle:android (*@\textbf{OR}@*) intitle:"mobile app")
```

Listing 6.1 – Search query used for automated search of white literature

Application of Selection Criteria. The search results are then filtered in order to obtain an initial set of primary studies by applying a set of well-defined inclusion and exclusion criteria. In order to systematically control the quality of the primary studies, three sets of criteria are defined: one general (i.e., applying for both WL and GL), one specific to WL, and one specific to GL. A paper is included only if it satisfies all inclusion criteria and none of the exclusion ones.

In order to further ensure the quality of GL primary studies, a subset of 5 quality-evaluation criteria presented by Garousi et al. is adopted [228]. A 3-point Likert scale (yes=1, partly=0.5, and no=0) is used to assign the quality scores. A GL study is considered of sufficient quality if it scores at least 2 out of 5 total points. Table 6.1 reports the considered selection criteria.

As recommended in [73], two researchers inspect a random sample of the studies. For assessing the objectivity of this phase the inter-researcher agreement is measured, achieving a substantial agreement Cohen Kappa value of 0.79, falling slightly below the one recommended in [230], equal to 0.80.

The application of the criteria terminates with the inclusion of 6 WL primary studies and 16 GL primary studies.

Snowballing Process. Once the preliminary set of primary studies is defined, we conduct a snowballing process. Regarding WL, we adopt a standard iterative backward and forward snowballing process [229]. During this process, 629 potential relevant studies are analyzed, leading to the inclusion of 1 additional primary study. For the GL, due to the high volume of primary studies to be considered, we limit the snowballing to the links referenced in the GL primary studies (i.e., we do not consider backward links), and stop the process after the first iteration. A totality of 16 new primary studies is included through this snowballing process.

Table 6.1 – Selection criteria for the multivocal literature review

Type	Description
General-Inclusion	Studies focusing / software architecture
General-Inclusion	Studies focusing on design or development of Android apps
General-Exclusion	Studies not published in English
General-Exclusion	Duplicate or extensions of already included studies
General-Exclusion	Studies which are not available
General-Exclusion	Studies not focusing on native Android applications, e.g., Unity-based videogames, web-based apps
WL-Exclusion	Secondary or tertiary studies
WL-Exclusion	Studies in the form of editorials, tutorials, books, etc.
WL-Exclusion	Studies which have not been peer reviewed
GL-Exclusion	Studies reporting exclusively the basic principles about the Android platform and its architecture
GL-Exclusion	Studies reporting exclusively abstract best practices
GL-Exclusion	Studies reporting only trivial Android implementations
GL-Exclusion	Studies reporting an implementation without a discussion of its benefits and/or drawbacks
GL-Exclusion	Studies written for promotional purposes
GL-Exclusion	White literature
GL-Exclusion	Videos, webinars, etc.
GL-Quality	Is the publishing organization or the author reputable?
GL-Quality	Has the author published other studies in the field?
GL-Quality	Does the study add value to the research?
GL-Quality	Is the presentation of the study of high quality?
GL-Quality	Is the study supported by evidence, e.g., examples/data?

6.3.3.2 Practitioners Interviews

In order to complement the data extracted from the MLR, we conduct semi-structured interviews with Android practitioners. This step consists in designing a survey based on our research questions, contacting potential interviewees, conducting the semi-structured interviews, and post-processing the interview results. Interviewees are selected via convenience sampling by exploiting our collaboration network. Developers are required to be affiliated to different companies, belong to distinct business domains, and possess at least 5 years of Android development experience. Out of 18 contacted practitioners, 5 result to be available for the interview. Interviews range between 30 and 60 minutes.

6.3.3.3 Data Processing and Extraction

After the collection of the WL/GL primary studies and survey results, referred jointly as *data points*, the resulting 44 data points are uniformed and merged into a single pool. Subsequently, we extract from the data points the information necessary in order to answer our research questions. The data necessary to answer $RQ5.1$ is extracted by inspecting the data points, and subsequently identifying which architectural patterns and libraries are considered. In order to extract architectural practices ($RQ5.2$) we conduct iterative content analysis sessions [99] involving two authors of the paper. Finally, to answer $RQ5.3$, the data points are inspected to extract which QRs are deemed to be impacted by architectural decisions. The entirety of the data extracted is mapped to the originating data points for the sake of backward traceability and replication purposes.

6.3.3.4 Data Synthesis

Finally, the extracted data is processed and synthesized in order to answer our research questions. In order to answer $RQ5.1$ and $RQ5.3$ the extracted data can be processed quantitatively. Differently, in order to answer $RQ5.2$, we carry out a keywording process [80]. This process consists of grouping the architectural practices extracted from the data points according to their semantic similarity. This is achieved by labelling the single practices with representative keywords. The process is iterated by refining the keywords, till the grouped practices can be merged into a single guideline. Practices mapped to a guideline in this fashion are referred to as *supporting practices*. The first round of keywording leads to the identification of the general themes considered, while the subsequent ones to the formulation of the

guidelines. Figure 6.4 summarizes the relationships between the elements of the resulting catalogue of guidelines.

For more details on the research method, research execution, and extracted data, we refer the reader to the replication package of this study[10]. The package is made available with the aim of supporting independent verification and replication. It contains (i) the rigorous research protocol defined *a priori* which we follow, (ii) the entirety of the search and selection execution data, (iii) the raw data extracted from the data points, and (iv) the documentation of data analysis processes accompanied by the relative results.

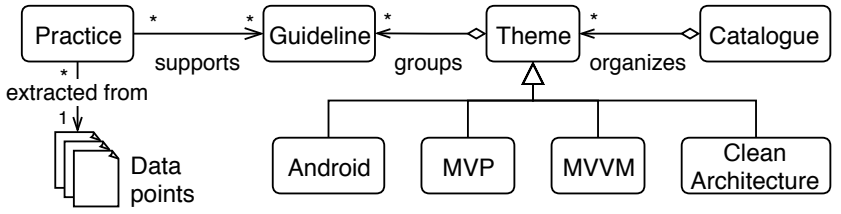


Figure 6.4 – Relationship between the elements of the catalogue of guidelines

6.3.4 Results

6.3.4.1 RQ5.1: Characteristics of Architecting Android Apps

Android architectural patterns (RQ5.1.1) The inspection of the extracted data lead to the identification of 7 architectural patterns considered when developing Android apps.
As shown in Figure 6.5, the most recurrent pattern results to be Model-View-Presenter (MVP), which is reported in 18 data points. Model-View-ViewModel (MVVM) results the second most frequent pattern. This could be associated to the recent introduction by Google of the `ViewModel` architectural component[11]. Due to its potentially drastic impact on Android architecture development, we expect the MVVM pattern to experience a fast growing trend of adoption in the coming years. Clean architecture principles [231] appear also to be frequently considered in the context of architect-

[10] https://github.com/AndroidGuidelines/ReplicationPackage
[11] https://android-developers.googleblog.com/2017/05/android-and-architecture.html

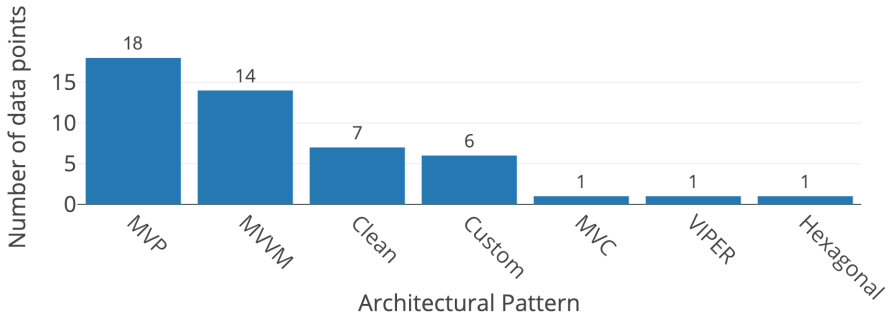


Figure 6.5 – Overview of architectural pattern recurrence

ing Android apps. Some data points also report *ad-hoc* custom solutions, such as MVP extended through manager classes, message-driven architectures, and architectures heavily relying on RxJava. Nevertheless, such architectural solutions appear to be fragmented and slightly less popular. Other architectural patterns, i.e.„ Model-View-Controller (MVC), View-Interactor-Presenter-Entity-Router (VIPER) [232], and hexagonal architecture [233], result to be only scarcely considered.

Findings for RQ5.1.1 (Android Architecture patterns): MVP is the most considered pattern, followed by MVVM. Clean architecture principles applied to Android apps are also frequently discussed. Heterogeneous *ad-hoc* solutions are also considered.

Libraries considered while architecting apps (RQ5.1.2)

An overview of the most recurrent libraries referenced when discussing Android architecture are reported in Figure 6.6.
Not surprisingly, RxJava[12] is the most mentioned library. RxJava enables a crucial programming paradigm for mobile apps, namely reactive programming. By adopting reactive programming, it is possible to efficiently deal with concurrency and asynchronous tasks, which are inherent to the mobile context. The second most recurrent library is Dagger[13], a framework maintained by Google which implements

[12] https://github.com/ReactiveX/RxJava
[13] https://google.github.io/dagger/

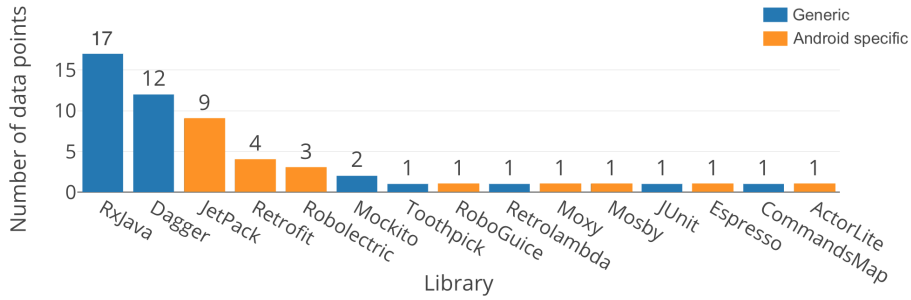


Figure 6.6 – Overview of library recurrence

the dependency injection pattern. This library constitutes a popular choice in order to manage dependencies, and potentially avoids unnecessary boilerplate code. JetPack, a recently released official Android library focusing on architectural components, is less popular. While the architectural relevance of such library in the Android ecosystem is clear, the lower occurrence of such library can be attributed to its recent release, and hence to the time required for its adoption. The other referenced libraries, like Retrofit, Robolectric, Mockito, are less recurrent, potentially due to their lower architectural relevance, hence pointing to a well scoped selection of data points. Interestingly, only 8 out of the 15 libraries reported are explicitly conceived for Android (see Figure 6.6). This shows that the Android architecture is “open", i.e., influenced by many generic libraries. In addition, of the Android specific libraries, only a few focus on architecture (JetPack, Moxy[14], and Mosby[15]).

Findings for RQ5.1.2 (Architecturally relevant libraries): RxJava is the most referenced library, followed by Dagger, JetPack, and Retrofit. Approximately half libraries are Android-specific, while only few focus specifically on Android architecture.

[14]https://github.com/Arello-Mobile/Moxy
[15]https://github.com/sockeqwe/mosby

6.3.4.2 RQ5.2: Android Architecture Guidelines

In order to answer RQ5.2, a total of 212 architectural practices are extracted from the selected data points. The first round of keywording leads to the classification of the practices into four emergent themes: *general Android architecture*, *MVP*, *MVVM*, and *Clean Architecture*. By applying recursively the keywording process, the practices are further clustered, till the synthesis of 42 architectural guidelines, reported in Table 6.2. As proxy for maturity of the guidelines, we utilize the number of supporting practices (SP) of each guideline. For the sake of conciseness, in the remainder of this section, we discuss the top-5 guidelines for each theme. The complete description of the entirety of the guidelines, supported by concrete source code examples, is available at a dedicated website available online[16].

General Android Architecture Guidelines
The first and most recurrent theme regards generic architectural practices for Android, i.e., not specific to any particular architectural pattern. In total, 120 practices are collected and synthesized into the following 15 general Android architecture guidelines reported in Table 6.2 and further detailed below.

A-1: *"Decouple components and explicitly inject/manage the dependencies among them"*.
While not strictly necessary, utilizing a dependency injection framework can drastically simplify the management of dependencies between Android architectural components. This supports a clean decoupling of architectural components and avoids unnecessary boilerplate for connecting them. Doing so not only improves the maintainability of the app, but also improves its testability by providing the possibility to inject mock implementations. The Dagger framework is commonly recommended to inject dependencies and solve problems afflicting reflection-based solutions.

A-2: *"Design components as independent entities as possible, build them around the features of the app and make them Android-independent"*.
As also remarked by two interviewees, a recurrent problem arises when common functionalities are not provided in base classes. This often leads to duplicated code, reducing the maintainability and testability of the app. Ideally, components should be independent from each other and their business logic should be clear and explicitly separated. By quoting one of the data points "your architecture should scream the purpose of the app". Decoupled components make it easier to focus on app functionalities and their issues, without dealing with bloatware. Additionally, this

[16] https://androidarchitectureguidelines.github.io/

Table 6.2 – Generic guidelines for architecting Android apps

Generic Android Architectural Guidelines		
ID	**#SP**	**Architectural guideline**
A-1	18	Decouple components and explicitly inject/manage the dependencies among them.
A-2	17	Design components to be as independent as possible, build them around the features of the app and make them Android-independent.
A-3	16	Counter the tendency of Activities to grow too big in size due to functionality/responsibility bloat.
A-4	14	Strive towards separation of concerns in your architecture, where each component has well defined responsibility boundaries, a purpose, (set of) functionality, and configuration.
A-5	10	When starting a new project, carefully select a fitting architectural pattern to adhere to.
A-6	8	Organize your Java/Kotlin packages and files either by layer or by app feature.
A-7	7	Take full advantage of libraries. Do not try to reinvent the wheel and loose time by implementing boilerplate code. Focus on what makes your app stand out from the rest and delegate what is left to libraries.
A-8	7	Locally cache data for supporting offline-first experience.
A-9	6	Use exclusively interfaces to let app modules communicate. This protects the architectural structure and helps defining a clear responsibility of modules.
A-10	5	Avoid nested callbacks, as they could lead to a "callback hell". Approximatively, more than 2 levels of callbacks are considered to reduce maintainability and understandability. This problem is commonly fixed by taking advantage of the RxJava library.
A-11	4	Employ well-defined and accepted coding standards, as they improve both code understandability and maintainability.
A-12	3	Use a dedicated module to persist as much relevant data as possible. This data source should be the single source of truth driving the UI.
A-13	3	Take into consideration the lifecycle of Android components (e.g., Activities and Services) – also with respect to other components – and design them as short-lived entities.
A-14	1	Have special care in designing background tasks, especially by considering the apps' lifecycle.
A-15	1	Use permissions consistently. Every component of an app that has a permission must be declared also at the app level.

ID = guideline identifier, #SP = number of supporting practices.

enables a higher testability of the core logic of the app by making components unit-testable (ideally without requiring an emulator). Finally, by decoupling the business logic from frameworks, more emphasis is put on the business logic, making an app more testable, maintainable, and of low technical debt.

A-3: *"Counter the tendency of Activities to grow too big in size due to functionality/responsibility bloat"*. Android Activitiesshould ideally contain exclusively logic handling the user interface (UI) and operating system interactions. Nevertheless, a common architectural issue consists of delegating too many functionalities and responsibilities to a single Activity. This leads to Activities slowly becoming god-classes. As the Android framework does not support the reuse of methods implemented in activities, code tends to be directly copied into other ones, increasing code duplication and impacting negatively the app's maintainability. Additionally, testing might become a challenging task, as complex business logic could reside in Activities, which by themselves result arduous to unit test. Finally, as activities are kept in memory at runtime, "god-activities" can lead to the deterioration of apps' performance.

A-4: *"Strive towards separation of concerns in your architecture, where each component has well defined responsibility boundaries, a purpose, (set of) functionality, and configuration"*.
Architectural components of an app should have a single, well defined, responsibility. As a component grows bigger, it should be split up. By following the single responsibility principle, the app architecture naturally supports the structure of developer teams and development stages. Additionally, monoliths are detected in the early stages and modules become testable in isolation. Finally, if the app is built using Gradle, modularization can improve the performance of the build process and ease the development of Instant apps[17]. It is important to notice that, while modularization may imply little effort if considered early in the project, it might become an extremely expensive process in later development stages.

A-5: *"When starting a new project, carefully select a fitting architectural pattern to adhere to"*.
Picking the right architectural pattern (e.g., MVP or MVVM) for the context and business goals of the app is a crucial decision. By adhering to an architectural patter selected *a priori*, separating responsibilities into components becomes a more straightforward process, and the growth of architectural technical debt is hindered. It is important to note that, when a certain level of adaptability/maintainability is not required, the selection of an ill-suited architectural pattern might lead to

[17] https://developer.android.com/topic/google-play-instant

over-engineering. Choosing the architectural pattern to adopt is hence a non-trivial decision which should be taken by considering the context of apps, and their business/organizational goals.

MVP-specific Android Architecture Guidelines
The second most considered theme regards practices related to the MVP architectural

Table 6.3 – MVP-specific guidelines for architecting Android apps

MVP-specific Architectural guidelines		
ID	**#SP**	**Architectural guideline**
MVP-1	9	Provide Views with data which is ready to be displayed.
MVP-2	5	Presenters should be Android- and framework-independent.
MVP-3	5	Access (and cache) the data provided by Models via app-scoped dedicated components.
MVP-4	4	Clearly define contracts between Views and Presenters.
MVP-5	4	The lifecycle of Presenters should follow the lifecycle of the Views, but not by replicating the complexity of the lifecycles of Android components.
MVP-6	3	Avoid to delegate too many responsibilities to Presenters, as they have the tendency to become bloat classes.
MVP-7	2	Make Presenters dependent on Views, and not Activities.
MVP-8	2	Views are passive and should always manage and expose only their state.
MVP-9	2	Strive towards putting as much of the app's business logic as possible in Presenters.
MVP-10	2	Inject dependencies to Presenters into the Views when instantiating the Presenters, as this reduces coupling issues and null checks.
MVP-11	1	If an app has multiple Presenters, do not let them communicate with each other.
MVP-12	1	If necessary, retain fragments for avoiding memory leaks due to configuration changes in the activities.

ID = guideline identifier, #SP = number of supporting practices.

pattern. In total, 40 practices are collected and synthesized in the 12 MVP-specific guidelines reported in Table 6.3 and further detailed below.

MVP-1: *"Provide Views with data which is ready to be displayed"*.
The view layer of Android apps tends to become bloated with responsibilities, and hence becomes harder to maintain. In order to alleviate such problem, Activities and Fragments can be provided with preprocessed data ready to be displayed. This can be achieved by delegating data-processing tasks to one or more dedicated components. In such manner, Activities and Fragmentsare relieved from the task of transforming and filtering domain-specific data, potentially improving the testability and usability of the app.

MVP-2: *"Presenters should be Android- and framework-independent."*.
To abstract Presenter components from the implementation details, Presenters should ideally avoid dependencies to the Android framework. This also entails not creating a lifecycle in Presenters, as it may hinder their maintainability and evolvability. In order to access app resources and preferences, View and Model components can be used instead, respectively. Additionally, by developing Presenters as only dependent on Java, the testability of Presenters drastically improves, as now non-instrumented unit-test cases can be written for such components.

MVP-3: *"Access (and cache) the data provided by Models via app-scoped dedicated components"*.
When developing an Android app, a common issue which might emerge is related to restoring the state of Views. This issue can be solved by adapting slightly the architecture of apps. Specifically a data manager component (e.g., a data store or Jetpack Repository) can be introduced. This component is responsible for data related tasks such as fetching data from the network, caching results or returning already cached data. By scoping such component at the app level and not at the one of single Activities, issues relative to restoring View states, e.g., in the occurrence of a screen orientation change, are solved through an architecturally maintainable solution.

MVP-4: *"Clearly define contracts between the Views and the Presenters"*.
Before starting to develop a new app feature, a good architectural practice consists in writing a contract documenting the communication between the View and the Presenter. The contract should document for each event in the View which is the corresponding action in the Presenter. By implementing contract interface classes, the source code of apps become more understandable, as the relation between the View and the Presenter is explicitly documented.

MVP-5: *"The lifecycle of Presenters should follow the lifecycle of the Views, but not by replicating the complexity of the lifecycles of Android components"*.
By having callbacks related to the Activity lifecycle in Presenters, Presenters become tightly coupled to Activities lifecycle. This can have a negative impact in terms of maintainability. From an architectural perspective, Presenters should not be responsible for data-related tasks. It is hence advised not to retain Presenters. An alternative solution would be to use a caching mechanism to retain data, keep Presenters stateless, and destroy Presenters when their corresponding Views are destroyed.

MVVM-specific Android Architecture Guidelines
Another recurrent theme identified in the practices regards the MVVM pattern. In total, 24 practices are collected and synthesized in the 10 MVP-specific guidelines

Table 6.4 – MVVM-specific guidelines for architecting Android apps

MVVM-specific Architectural Guidelines		
ID	**#SP**	**Architectural guideline**
MVVM-1	5	Models, Views, and ViewModels should exclusively expose their state instead of state and data separately.
MVVM-2	4	The app should possess a single source of truth of data.
MVVM-3	3	Models should be evolvable/testable independently from the rest of the app.
MVVM-4	3	ViewModels should not refer to View-specific components.
MVVM-5	2	Views should always know about changes after ViewModels, no matter how trivial an operation may be.
MVVM-6	2	Adopt one Model for each feature of the app.
MVVM-7	2	Keep ViewModels as simple as possible. When needed, transfer responsibility to other layers, e.g., Models or other components such as data transformers, components factories, etc.
MVVM-8	1	The state of the app should be defined in the Models only, whereas Views and ViewModels should be stateless.
MVVM-9	1	The data produced by the Models should be reliable and of high quality.
MVVM-10	1	Networking or data access functionalities should be performed exclusively by Models.

ID = guideline identifier, #SP = number of supporting practices.

reported in Table 6.4 and further detailed below.

MVVM-1: *"Models, Views, and ViewModels should exclusively expose their state instead of events or data separately"*.
For example, to ensure that Views display up-to-date content, it is recommended that ViewModels expose states rather than just events. This can be achieved by bundling together the data that needs to be displayed. In such way, when one of the fields to be displayed changes, a new state is emitted and the View is updated. This entails that each user interaction involves an action in the ViewModel, enabling a clean separation of concerns between MVVM components.

MVVM-2: *"The app should possess a single source of truth of data"*.
In the context of mobile applications, consistency of data can become an issue. While caching mechanisms allow to save energy and bandwidth, multiple data sources can create inconsistencies and even conflicting Views. In order to avoid such issues, it is recommended to designate a dedicated component as single source of truth for the entire app. Specifically, in the context of MVVM, the Room persistence library[18] is an official architectural component of Android which is specifically tailored for such task.

MVVM-3: *"Models should be evolvable/testable independently from the rest of the app"*.
Well-designed ViewModels should completely decouple Views from Model classes. In such way, by strictly adhering to the MVVM pattern, Models and Views can evolve independently and be tested with ease. Additionally, by applying the inversion of control principle and implementing ViewModels decoupled from the Android framework, it is possible to test ViewModels via unit tests. In contrast, if the binding between the MVVM components is too complex and intertwined, testing and debugging Android apps can become a cumbersome challenge.

MVVM-4: *"ViewModels should not refer to View-specific components"*.
Passing context to ViewModel instances can result in a dangerous practice. In fact by storing the reference to an Activity in a ViewModel, once the Activity gets destroyed (e.g., due to a screen rotation), a memory leak could occur. By quoting a Google Android Developer Advocate: "*The consumer of the data should know about the producer, but the producer - the ViewModel - doesn't know, and doesn't care, who consumes the data.*"[19]. In order to adhere to this guideline, the LiveData[20] architectural class

[18] https://developer.android.com/jetpack/arch/room

[19] https://medium.com/upday-devs/android-architecture-patterns-part-3-model-view-viewmodel-e7eeee76b73b

[20] https://developer.android.com/topic/libraries/architecture/livedata

provided by the Jetpack library can be used, so that Activities can simply observe the changes of the ViewModel's data.

MVVM-5: *"Views should always know about changes after ViewModels, no matter how trivial an operation may be"*.
Adhering to this guideline implies that all the logic in the Views should be moved to the ViewModels. While the purpose of ViewModels is to pre-process data to be ready to use by Views, it might be tempting to implement minor operations in Views. Nevertheless, adhering to this guideline guarantees a higher level of consistency and reliability of all the components which are based on the ViewModels.

Clean Architecture Android Guidelines

Table 6.5 – Clean architecture guidelines for architecting Android apps

Clean Architecture Architectural Guidelines		
ID	**#SP**	**Architectural guideline**
CLEAN-1	13	Business logic should be completely decoupled from the Android framework.
CLEAN-2	5	The outer architectural layer should contain the entirety of the app's UI components.
CLEAN-3	4	The framework and devices layer should include the entirety of the app components which depend on Android.
CLEAN-4	4	Each architectural layer should possess its own data model.
CLEAN-5	2	Keep the UI thread as lightweight and isolated as possible.

ID = guideline identifier, #SP = number of supporting practices.

While rather a set of architectural practices than a pattern, clean architecture results to be a common theme in Android architecture literature. In total, 28 practices on this theme are collected and synthesized into the 5 architectural guidelines reported in Table 6.5 and further detailed below.

CLEAN-1: *"Business logic should be completely decoupled from the Android framework"*.
By adhering to the clean architecture principles, the innermost layers of an app (i.e., where all the business logic of the app resides) should be "frontend agnostic". This means that this layers are completely decoupled from the Android framework, and could be ideally implemented as pure Java packages. Additionally, as this layers represent the core of Android apps, they should be developed before all other layers.

Changes to the innermost layers should be driven exclusively by business decisions.

CLEAN-2: *"The outer architectural layer should contain the entirety of the app's UI components"*.
In order to ensure a clear separation of concerns among the clean architecture layers of an app, it is paramount that everything related to Android UI is grouped in a module residing in the outer architectural layer. As the other architectural layers of the app should be "frontend agnostic" (see guideline CLEAN-1), different patterns (e.g., MVP or MVVM) can be implemented in this layer. Activities and Fragments should not handle any other logic than the one necessary to render the UI. This allows a clear separation of concern among the architectural layers of an app, enhancing the understandability, modifiability and testability of its components.

CLEAN-3: *"The framework and devices layer should include the entirety of the app components which depend on Android"*.
In accordance to the clean architecture principles, all components related to the framework should be grouped in the outer architectural layer. This includes all components which contain Android specific implementations, which should not be present in the business logic layers. Examples include, in addition to the user interface model, the data persistence module (e.g., LiveData, DAOs, ORMs, Shared Preferences, Retrofit, etc.) and eventual dependency injection frameworks.

CLEAN-4: *"Each architectural layer should possess its own data model"*.
By implementing a data model at every layer, a high degree of decoupling between layers can be achieved. Specifically, by following this guideline, the outer layers of apps can be implemented without any explicit knowledge of the implementation details of the inner layers. This means that the origin of the data becomes transparent to the client and hence, in a repository pattern fashion, data sources can be added, removed, or changed without much effort.

CLEAN-5: *"Keep the UI thread as lightweight and isolated as possible"*.
In accordance to guidelines CLEAN-1 and CLEAN-2, presenters residing in the outer layers of apps modeled through clean architecture principles should be kept lightweight. In fact, Presenters should be composed with interactor components, i.e., use cases residing in the business logic layers, which are responsible for executing tasks outside the main UI thread of the app. Once the task are finished, the Views are updated through a callback with the processed data. Besides callback-based communication among components, other techniques used in order to keep the UI thread lightweight rely on the inversion of control principle and intent-based communication.

Throughout this study we notice that low-level technical concerns (e.g., management of screen rotation, access to sensors) are often intertwined with architectural concerns (e.g., how to structure the whole app, how UI events should flow within the app) without a clear separation between them. Our emerging guidelines can help Android developers in starting to (i) abstract from the low-level details of the app, and (ii) reason on its overall structure and related architectural concerns.

Finally, it must be noted that the identified guidelines should be seen as recommendations, rather than strict rules to be followed at any cost. Indeed, by quoting one of our data points: *"No architecture is perfect for every use-case and Architecture Guidelines are just recommendations. Hence [developer] feel free to use whatever suites your use-case."*. In other words, we advise developers not to consider the guidelines provided in Table 6.2 as a whole, but rather to reflect carefully on which ones should be applied in their projects, depending on the current technical, business, and organizational context.

Findings for RQ5.2 (Android Architectural Guidelines): 212 architectural practices are extracted and synthesized into 42 architectural guidelines, reported in Table 6.2. Four main overarching themes emerge from the guidelines: *generic Android architecture guidelines*, *MVP-specific*, *MVVM-specific*, and *Clean Architecture*. The 5 most mature guidelines per theme are detailed, while the remaining are documented in Table 6.2.

6.3.4.3 RQ5.3: Quality Requirements

Figure 6.7 shows the QRs considered when architecting Android apps. We observe that maintainability, testability, and performance are the highest ranking ones.

Overall, the results gathered for RQ5.3 are in line with what is intuitively evinced by inspecting the results of RQ5.2. In fact, a high number of guidelines deal with modularization, separation of concerns, and deal with components size coupling, etc. The application of such principles impacts primarily maintainability and testability, as discussed in Section 6.3.4.2. Interestingly, those principles are also strongly related to the maintainability issues frequently occurring over the lifetime of Android apps [109]. Additionally, guidelines such as avoiding "god-activities" (A-3) or use specific data management solutions (MVP-3) can drastically impact the performance of apps. Architectural guidelines influencing primarily other QRs are overall less frequent.

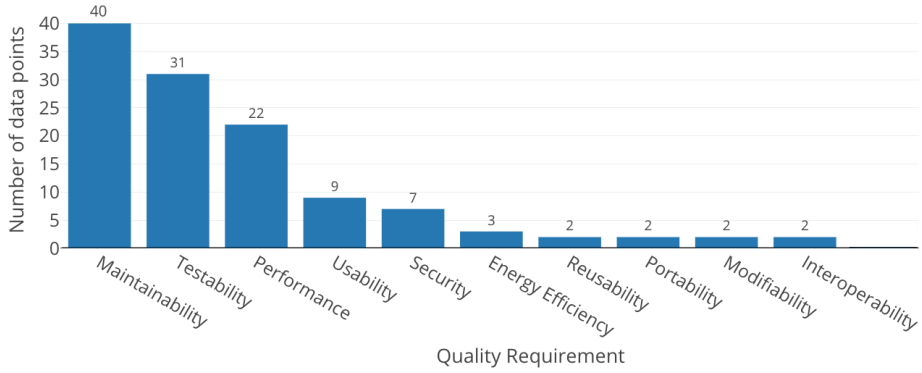


Figure 6.7 – Quality Requirements considered while architecting Android apps

Moreover, we observe that most quality requirements regard development-time QRs (e.g., maintainability, testability) rather than runtime attributes (e.g., performance, energy). The focus on static QRs in Android emerges also from the guidelines reported in Section 6.3.4.2. In fact, most of the synthetized guidelines consider development-time architectural views of apps (e.g., A-1) rather than runtime ones (e.g., A-8).

Findings for RQ5.3 (Android Architecture Quality Requirements): Maintainability, testability, and performance are the most considered QRs when architecting Android apps. Most QRs regard development-time attributes.

6.3.5 Threats to Validity

External Validity. The primary threat to this category is represented by the selection of the data points, which might not be representative of the state of the art and practice. To mitigate this threat, we adopt 3 different data sources (semi-structured interviews, WL, and GL). This leads to a more heterogeneous set of data points for our study. Additionally, to ensure the soundness and quality of the MLR data points, a thorough selection and quality evaluation process is conducted via a set of well-defined evaluation criteria. To identify the interviewees, convenience sampling is adopted. This constitutes a threat to external validity, mitigated by selecting inter-

viewees that resulted heterogeneous in terms of type of apps developed, company, background and developer role.

Internal Validity. To mitigate potential threats to internal validity, we follow a rigorous research protocol defined *a priori*. To avoid biases related to data collection through semi-structured interviews, we perform such step prior to the MLR execution and by following an interview guide as part of the protocol. Internal validity threats of the MLR are mitigated by following established guidelines for conducting WL reviews [227, 74] integrated with guidelines for the inclusion of GL [228].

Construct Validity. The most prominent threat to construct validity regards the potential inappropriateness of data point selection. To mitigate this treat we use multiple data sources. As suggested by Wholin et al. [73], the quality of the MLR selection process is ensured by measuring inter-researcher agreement on a random subsample of potentially relevant studies. Additionally: (i) we perform the MLR by adhering to well-documented search and selection processes predefined in a rigorous protocol, and (ii) the semi-structured interviews are conducted exclusively with developers with at least 5 years of experience. The adoption of Google Scholar and Google Search Engine might constitute a bias due to their underlying algorithms. We mitigate this threat by using well-defined selection criteria and conducting a snowballing process.

Conclusion validity. The data extraction and synthesis processes are conducted by strictly adhering to the *a priori* defined protocol, designed specifically to collect the data necessary to answer our RQs. This reduces potential biases associated to such processes and guarantees that the extracted data is appropriate for our RQs. Furthermore, the study was conducted by adhering to best practices from several sources [73, 227, 74, 228, 229]. We document each phase of our study in a publicly available research protocol, thus aiding replicability. To ensure the quality of the guidelines, the keywording process and guideline synthesis is conducted collaboratively by two researchers. Conflicts are managed with the intervention of a third researcher. Possible threats related to the interview process are mitigated by conducting internal and external pilots during the interview design phase. This is repeated several times to extensively refine the interview process. Another potential threat to conclusion validity is entailed by the low number of participants. This threat has been mitigated by adopting a data fusion approach, i.e., treating as homogeneous data points the information collected with the MMS, independently of their original data source. This allowed us to treat equally the data originated via the different data sources, and hence do not put particular emphasis, weight, or special treatment, to

any specific data sources.

6.3.6 Related Work

Despite the wide diffusion of Android apps and their increasing complexity [234], at the time of writing we found a surprising low number of **research studies about the architecture of mobile apps**. By mining and reverse engineering the architecture of more than 1,400 Android apps, Bagheri et al. studied the role of software architecture in the design and development of mobile software, extracted the architectural principles that have been applied by app developers, and identified architectural anti-patterns of the Android programming model [221]. They found that Android apps are complex, composed of tens of components, and organized according to multiple architectural styles. These findings motivated us to investigate how developers architect Android apps, eventually leading to the results in this study. Even though our study and that of Bagheri et al. share the same target audience, i.e., Android developers and researchers, the research goals are profoundly different: Bagheri et al. focus on known architectural principles and how they are reflected in the Android programming model, whereas we aim at characterizing the state of the art and practice on architecting Android apps. Moreover, the methodologies applied in the two studies are completely different – Bagheri et al. mined the apps from app stores and statically analyzed their bytecode, whereas we contact Android practitioners complemented with a systematic analysis of the grey and white literature.

An exploratory study targeting common architectural characteristics of 12 real Android applications is reported in [235]. The study is based on the partial extraction of the architecture of the apps using the JDepend tool, followed by the manual analysis of the source code of the targeted apps. The main results of such manual analysis revealed that MVC is a recurrent pattern in Android (although with some violations). In our work we apply a totally different research methodology, where we target professional developers working on industrial projects, rather than developers working on open-source apps. Also, the work proposed in [235] is exploratory in nature and aims at observing the characteristics of the architecture of Android apps, whereas we aim at providing actionable guidelines for helping developers during their everyday activities.

A new MVC-based architectural pattern called Android Passive MVC, is proposed in [236], with the aim of producing Android apps with better maintainability, extensibility, performance, and less complexity. The proposed pattern has been applied to

an example of social networking app in collaboration with a development company. Differently from [236], we do not aim at providing a new architectural pattern, rather we accept the existence of many pre-existing ones in Android apps (also confirmed in [221, 235]) and aim at supporting developers while architecting Android apps, without forcing them to learn and apply new (potentially unsupported) architectural patterns.

The authors of [237] performed a preliminary study on how to develop Android apps according to the Software Product Line (SPL) approach. Their case study shows that an adaptation of the principles for SPLs can be adopted for developing Android apps. We differ as we do not imply a change in developers' habits by means of a new development paradigm like SPL, but aim at supporting them in taking better-informed decisions about the architecture of their apps.

A study about the challenges faced by mobile app developers (not only Android) has been proposed by Joorabchi et al. [238]. The challenges have been extracted in a qualitative manner from a combination of 12 interviews with practitioners and an online questionnaire with 188 participants. Differently from us, they focus on mobile apps in general (incl. web and hybrid apps) and are orthogonal to architectural concerns, i.e., they do not cover the challenges directly related to architecture, but focus on challenges related to e.g., testing. Interestingly, in our study we found some confirmation of their insights, e.g., the importance of testability, partially managed by following the MVP or MVVM patterns.

From a methodological perspective, **multivocal studies** (i.e., systematic studies targeting both grey and white literature) are being published only recently in the field of software engineering [226], e.g., investigating code smells in testing artifacts [239], the startups ecosystem [240], and microservices [241]. Researchers are also complementing multivocal studies with other research methodologies (e.g., semi-structured interviews in our case), thus leading to mixed-method studies. For example, Maro et al. combined a tertiary literature review, a case study with a company, and a multivocal literature review to identify challenges and solutions about software traceability in the automotive domain [242]. In the literature there is no multivocal study on mobile apps. So, even though also our research combines a multivocal study with other research methodologies (e.g., interviews), the subjects and therefore the outcomes of our study are novel by focussing on architecting practices for Android apps.

6.3.7 Conclusions and Future Work

This chapter presents a mixed-method empirical study on the state of the art and practice on architecting Android apps. A key result of our study is the set of 42 evidence-based guidelines for architecting Android apps. In addition to the guidelines, our study reveals that: (i) so far there are very few academic articles targeting Android architecture (this may be an emerging research gap for academic researchers), (ii) MVP and MVVM are the most recurring architectural patterns, (iii) RxJava, Dagger, and JetPack are the most mentioned libraries when dealing with the architecture of Android apps, and (iv) maintainability, testability, and performance are the most considered QRs when architecting Android apps.

Our results provide developers with an organized set of guidelines for taking action towards improving the architecture of their apps (e.g., when adopting MVP, strive towards making Presenters Android- and frameworks-independent – cf. MVP-2). Researchers, in turn, can benefit from the provided overview of the state of the art and practice, e.g., for tailoring their research towards those QRs that concern developers the most when dealing with the architecture of their apps.

This study opens for many future research directions. Firstly, we are planning a large-scale confirmatory study involving practitioners for checking the correctness and completeness of the proposed guidelines. Secondly, properly designed analysis tools might automatically check violations of guidelines via static and/or dynamic analysis techniques, and recommend solutions for those violations. Furthermore, we strive towards the implementation of an Android reference architecture [52], in order to lay the foundations for compliance-based architectural technical debt analyses [1]. Thirdly, it would be interesting to empirically assess how applying the proposed guidelines can actually impact the quality of the mobile app, thus enabling developers to quantitatively assess the gains in having well-architected apps, potentially speeding up the industrial adoption of the proposed guidelines.

Conclusions Part III

7 Discussion

Contents

In this closing chapter we document (i) a revisitation of the research questions underlying this study, (ii) the encompassing threads to validity which could affect the presented results, (iii) an overview of the implications of our findings, and (iv) the closing remarks on future work and outlook implied by our investigation.

7.1 Research Questions Revisited

In this section, we discuss how our thesis answers the research questions presented in Section 1.6.

RQ1: What is the state of the art of architectural technical debt identification?

ATD identification is attracting a growing scientific interest in the latest years. The research landscape is diverse, with ATD-related studies presented at heterogeneous venues, including the International Conference of Software Engineering (ICSE), the Managing Technical Debt workshop (now TechDebt Conference), and the International Conference of Software Architecture (ICSA).

ATD identification is strongly rooted in TD identification techniques working at the source code level. This finding is most prominent if the abstraction level, input, and ATDI definitions of the approaches presented in the literature are considered. A large and heterogenous set of analysis types characterize the state-of-the-art of ATD identification techniques, ranging from the identification of architectural antipatterns, to dependency analysis, change impact analysis, and even manual classification of software artifacts. ATD resolution is considered only by a small portion of the literature, indicating promising directions for future research directions. Additionally, the temporal dimension is considered only by a small portion of state-of-the-art identification techniques. Regarding tool availability, while a large number of tools for ATD identification are being proposed in the literature, only a small portion of them is publicly available to date.

The vast majority of approaches presented in the literature are academic-only, highlighting the ongoing requirement of academia-industry collaboration to accelerate knowledge transfer and tuning the research focus on the most-critical industrial needs. Research rigor (in terms of reusable study designs) and industrial relevance (in terms of targeted industrial subjects and scale, and used methods) display that current ATD identification approaches are potentially ready for industrial adoption. However, current limitations in the literature regarding context description and discussion of the validity threats, could pose a potential risk to the successful adoption of the approaches in industrial contexts, as deployment characteristics, and potential drawbacks of the approaches, remain often to be discovered.

RQ2: How can design issues detectable by tools be used to gain an overview of the architectural technical debt residing in a software system?

In Chapter 3, we presented a multi-step approach designed to build an ATD index (*ATDx*), based on a set of source-code analysis rules. The index, conceived to leverage design issues detectable by tools, provides insights into a set of ATD dimensions building upon existing architectural rules by leveraging statistical analysis. We conducted an empirical evaluation of our approach by implementing an instance of *ATDx* based on a set of identified SonarQube architectural rules and two distinct open-source software portfolios, comprising a total of 237 software projects. Based on the feedback provided by 47 software practitioners, we demonstrated how ATDx can be used to gain an overview of the ATD present in software-intensive systems according to different ATD dimensions, and that the results are actionable to resolve ATD issues. The *ATDx* approach can be adopted by researchers and practitioners alike in order to gain a better understanding of the nature of the ATD present in software-intensive systems, and provides a systematic framework to implement concrete instances of *ATDx* according to specific project and organizational needs.

RQ3: What is architectural technical debt according to software practitioners?

With our investigation reported in Chapter 4, we established our theory of architectural technical debt, by grounding our findings in the knowledge of experienced software developers. From our study 8 prominent categories related to ATD phenomena emerged, which constitute the foundation of our theory on ATD, namely: *ATD items*, *causes*, *consequences*, *symptoms*, *management strategies*, *prioritization strategies*, and *people*, and *communication*.

We studied the relations of heterogeneous nature which connect the various categories of our theory, outlining an encompassing theory based on the data we gathered. In addition to the analysis of the relations between the emerging categories of our theory, we also conducted a deep-dive into each one of them, further characterizing our theory by studying the most prominent concepts of each of the emerging categories. Our theory demonstrates the complexity and multifaceted nature of ATD phenomena. Such finding underlines that, while some degree of simplification might be needed to identify ATD items, the holistic view of the context in which ATD emerges plays a crucial role for the comprehensive understanding of the nature of the ATD items.

RQ4: How are maintainability issues of Android applications characterized?

In our research reported in Chapter 5, we investigated the nature and evolution of Android issues. Specifically, we shed light on (i) how often various types of maintainability issues occur over the lifetime of Android apps, (ii) the evolution trends of the density of maintainability issues in Android apps, and (iii) an in-depth characterization of development activities related to maintainability hotspots. Among different types of maintainability issues, we discovered *code duplication* to be the overall most recurrent type, potentially due to the coding paradigm considered. Maintainability hotspots do not occur often in the lifetime of Android apps. Nevertheless, on average, each app experiences at least one maintainability hotspot. Additionally, independently from the type of development activity carried out, maintainability issues grow according to different trends until they stabilize. Nevertheless, maintainability issues in Android apps result never to be fully resolved.

RQ5: How can architectural technical debt items be identified in Android Applications?

In Chapter 6 we present an approach based on the establishment of an encompassing reference architecture for Android applications, followed by a subsequent compliance checking process. Specifically, our approach is composed of five main steps, namely (i) architectural guidelines extraction, (ii) establishment of an Android reference architecture, (iii) reverse engineering of implemented architecture, (iv) compliance checking, and (v) quantitative assessment of compliance violations (for a more comprehensive overview of the approach see Figure 6.1). Additionally, we executed the first step of our approach by studying, via a mixed-method empirical study, the current trends of architecting Android applications. This resulted in the synthesis of a set of 42 architectural guidelines for architecting Android apps, subdivided into 15 generic guidelines, 12 MVP-specific, 10 MVVM-specific, and 5 regarding clan architecture principles.

RQ: What strategies can be used to identify and manage architectural technical debt?

After answering all the sub-research questions of our thesis, we can address the main research question which is at the foundation of our investigation. From our findings

we can evince that source-code analysis of software-intensive systems is *de facto* the most recurrent methodology currently adopted for identifying architectural technical debt. We developed a methodology based on such result, presenting an architectural technical debt index, reported in Chapter 3. A similar analysis strategy was adopted to study architectural debt specific to Android applications, presented in Part II of the thesis. Nevertheless, as can be concluded from our theory on architectural debt reported in Chapter 4, statically detectable source-code debt is only one of the many facets through which architectural debt can be observed. This points to the necessity to study, conceive, and develop holistic approaches which consider the many dimensions which characterize the complex phenomenon of architectural technical debt. While the road towards the establishment of such approaches could be challenging, it is exactly what drives forward our research and future endeavors.

7.2 Threads to Validity

In this section we report general threats to validity which could affect the conclusions of our thesis. Such threads are to be intended as complementary w.r.t. the threads specific to each study reported in the previous chapters of this thesis.

7.2.1 External Validity

This threat refers to the generalizability of our results beyond the considered experimental settings, with particular emphasis in our case in the experimental subjects adopted. The results presented in this thesis do not claim extensive comprehensiveness or completeness, as the results are based on the analysis of a selection of experimental subjects according to rigorous selection criteria defined *a priori*. Regarding the studies based on source code analysis, e.g., Chapter 3 and Chapter 5, the results are relative to a specific programming language, namely Java. The choice to focus on such programming language is dictated by the availability of established tools covering such language (see Chapter 3), or by the extent in which it is used in specific software ecosystems (see Chapter 5- 6). It is hence crucial to consider that our results based on source code analysis presented in this thesis may differ if other programming languages and technologies are considered. In order to mitigate this and other threats related to subject selection, we gave highest priority throughout the entirety of our studies to rigorous, documented, and replicable, strategies for subject selection, as further detailed in the previous chapters.

7.2.2 Internal Validity

Despite our best efforts, the results reported in this thesis could be affected by threats to internal validity. In order to mitigate such threat, we consider paramount from an ethical and scientific standpoint to consistently share with each of our studies the entirety of the collected data, data gathering / analysis scripts, and extensive research protocol documentation. We believe the dissemination of replication packages to be a necessary step to accelerate scientific progress, while making minute and technical details available for review, correction, and adaptation for future researches. As exception, the data collection of the study reported in Section 4 relied on extensive interviews with software practitioners. As it is our priority to guarantee and protect the anonymity of our participants, their companies, and collaborators, we kept confidential all their identifying details. The verifiability of such study has hence to be evinced from the soundness of the followed approach, and the extensive use of direct quotes from participants (albeit excerpted).

7.2.3 Construct Validity

In order to mitigate potential threats to construct validity, we adopted as a standard procedure for the studies reported in this thesis the *a priori* definition of research protocols. This hinders the possibility to incur into fallacies dictated by data-driven modifications to the research methodologies adopted, and the potential underestimation of research process details. Additionally, the entirety of the research methods followed were designed via extensive discussions among at least three researches (either authors of the studies or external advisors), in order to mitigate possible inaccuracies which could have been introduced in a protocol by a single researcher. Another prominent threat is the focus on specific facets of ATD in each chapters (e.g., the focus on ATD identification via source code analysis of Chapter 2, or the focus on ATD detectable via compliance checking of Chapter 6). It is hence important to bear in mind that the results gathered in each study apply only to the specific facet of ATD considered, and do not claim to holistically focus on ATD (with the exception of the theory presented in Chapter 4).

7.2.4 Conclusion validity

A prominent threat to conclusion validity of the results reported in this thesis regards the completeness of our results. As reported in Chapter 4, architectural technical debt

is a multifaceted phenomenon, possessing various dimensions and influenced by numerous factors. While the research reported in this thesis is intended to progress our knowledge in architectural technical debt, the results do not claim completeness nor comprehensiveness w.r.t. the existing dimensions of architectural technical debt. As documented in [243], a temptation of both researchers and developers could be to "look for the magic wand", i.e., base the identification, analysis, and evaluation of architectural technical debt exclusively on source code analysis. Nevertheless, as reported in Chapter 1 and Chapter 4, architectural technical debt is a vast, encompassing, and only potentially marginally understood concept, which spans also over various non-source-code related concepts. Therefore it is paramount to interpret the results presented in this thesis in light of the specific research methodology, context, and scope under which each investigation was conducted. The specifics of each study frame the non-negligible viewpoint from which the vaster phenomenon of architectural technical debt was observed and studied.

7.3 Research Implications

In this section we provide a brief outline on how software researchers and practitioners can leverage the findings of our thesis to further progress the field of software engineering.

In Chapter 2 we present an outline of the current state of the art of architectural technical debt identification research. From an academic standpoint, such investigation aims at supporting researchers by concisely providing insights into the design of existing identification approaches, making it possible do adapt them or conceive new ones based on the current state of the art. From a practical point of view, by providing a characterization and comparison of existing ATD identification approaches, it is possible for practitioners to understand the capabilities of existing approaches, and potentially select the most fitting identification approach based on their requirements.

In Chapter 3 we present ATDx, an index to evaluate the ATD present in a software-intensive system. We base the calculation of ATDx on the prominent static source-code analysis tool SonarQube, which results to be widely adopted in industrial contexts[1]. The choice to adopt such static-analyzer is dictated, among other things, by the ease with which ATDx can be adopted in industry, supporting practitioners in an

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

eagle-eye evaluation of the ATD present in their software products. Additionally, we provide an in-depth documentation of the process we followed to establish ATDx, ranging from the selection of the metrics, their statistical analysis and manipulation, and aggregation. This provides researchers with the possibility to study the followed approach, making it possible adopt, modify, and improve it based on their specific requirement and context. Finally, we also make available a dataset of SonarQube rule violations[2], encompassing 237 projects for a total of more than 22.8K issues, on which future researches can be based.

Chapter 4 reports our theory of Architectural Technical Debt. Such theory is intended for both researchers and practitioners to get novel insight into the phenomena related to architectural technical debt. Our theory not only provides an encompassing description of the concepts related of ATD, but also reports data on which future research and practical solutions can be based. Interestingly, among other results, we identified a set of ATD symptoms, on which future ATD identification techniques can be based. On the same line, we elicited a set of ATD management and prioritization strategies, which provide novel insights on how ATD items are dealt with, giving practitioners the means to assess their current state of ATD management. In line with grounded theory principles, our theory is open to modification and extension, laying the groundwork for the establishment of a holistic overview of ATD according to new empirical data.

In Chapter 5 we document a study on the recurrence and evolution of maintainability issues in Android applications. Such study can support researchers investigating non-functional quality aspects of Android applications by providing empirical evidence on the recurrence and trends of different types of maintainability issues. Furthermore, in order to support Android developers, we leveraged our results to establish a set of development best practices, aimed at mitigating potential problematics related to software quality in which Android developers may incur.

Finally, in Chapter 6, we present a research approach aimed at the automatic detection of ATD hotspots in Android application. While the material reported in this chapter constitutes part of an ongoing research thread, the intermediate results already establish implications for future research and development activities. Specifically, as a first step of our approach, we synthesized a set of 42 architectural guidelines for Android applications, which can be leveraged by researchers and developers alike in order to prevent, analyze, and identify potential shortcomings of Android applications at the architectural level. Additionally, we also provide evidence of the current

[2]Including a filtered version according to the set of selection criteria used for our investigation.

state of the art of architecting Android applications in terms pattern used, supporting libraries, and non-functional properties considered. This lays the groundwork for a better understanding of architecting practices in the Android ecosystem, which recently gained interest from industry, but which result to be almost completely unexplored from an academic point of view.

7.4 Replicability

In the entirety of the studies presented in this thesis, we put our best effort to make the research processes and results as transparent as possible, in order to support independent scrutiny and replication of the work. In order to do so, each study is supported by a replication package, including the totality of source code used, data considered, and intermediate, additional, and final results. The only exception is for the study reported in Chapter 4, as the unprocessed data gathered could not be made public due to the human ethics guidelines governing the study (University of British Columbia Research Ethics Board, application number: H19-01125). In order to mitigate this potential shortcoming, we carefully documented in Section 4.2.2 the research method we followed throughout our investigation, and in the results section (Section 4.3) we referenced as much as possible direct quotes from our participants (albeit excerpted).

8 Conclusions, Future Work, and Outlook

This was a triumph
I'm making a note here
"huge success"
it's hard to overstate
my satisfaction

GLaDOS, *"Still Alive"*

The results of our study open numerous future research opportunities. This is highlighted even from first chapter of this thesis, where we identified a set of current ATD research trends and gaps. Specifically, it is interesting to notice that, even if architectural technical debt often requires time in order to become noticeable, only few studies currently consider the temporal dimension for ATD identification. Additionally, most approaches focus on a single, fine-grained, ATD symptom. This contrasts one prominent finding of our ATD theory presented in Chapter 4, where we show that a single ATD item can be traced back to a plethora of symptoms, e.g., performance, customer, and functionality issues. Interestingly, in Chapter 3, we present an index to detect code-related ATD via the measurement, analysis, and aggregation of architectural rule violations. The findings of such study, and the underlying methodology used, can be leveraged in order to create an index which goes beyond the analysis of statically-detectable source-code violations. Summarizing, the aforementioned findings highlight the potential which lies in combining the analyses of different types of symptoms, while considering their evolution, into a single unified viewpoint, in order to further progress, in terms of efficacy and effectiveness, automated ATD

identification and management techniques.

In addition to the establishment of what we envision could be an encompassing ATD index, the studies reported in this thesis offer the potential to be further refined and extended. Notably, the static source code analysis technique presented in Chapter 3 focuses on a limited subset of architectural rules, and considers exclusively Java-based experimental subjects. We aim at mitigating such threat to external validity in the future not only by reviewing the rule set and considering software projects based on other programming languages, but also by comparing our findings via the application of our methodology to other prominent analysis tools, such as CAST[1] and NDepend[2].

From a more theoretical standpoint, as future work, we strive towards the expansion and refinement of our ATD theory presented in Chapter 4, in order to further develop our theory and make it as encompassing and complete as possible. While we achieved theoretical saturation in our study, we set as one of our ultimate goals to continuously evolve and improve our theory as new data, knowledge, and findings on ATD become available. While we acknowledge the ambitiousness of such goal, we envision to tackle its feasibility by working in continuous incremental steps, and eventually establish an academic collaborative platform to keep track of new findings and the evolution of the theory to reflect the current state of the art and body of knowledge on ATD. In addition we envision to systematically map existing academic literature to elements of the theory, in order to uncover potential gaps in the current research activities, and identify interesting yet not studied research avenues related to ATD.

Finally, Part II of this thesis, which focusses on ATD specific to the Android ecosystem, presents a portion of our ongoing work targeting ATD specific to mobile software systems. Specifically, in this thesis we report the overview of our approach for ATD identification in Android application via compliance checking, followed by the implementation of the first out of five steps which such approach entails. We are spending ongoing research effort into the realization of the subsequent steps, following our vision to establish a light-weight, efficient, and effective approach to identify ATD hotspots in Android applications.

In conclusion, architectural technical debt is a multifaceted, complex, problem. A mischievous beast, manifesting itself as a subtle yet continuous cost, constantly slowing down development practices, and presenting an investment which, at the end

[1] https://www.castsoftware.com/
[2] https://www.ndepend.com/

of the game, is not worth taking on. Our ongoing research, and its underlying vision, aims to address the problem by conceiving a structured, holistic, methodology to identify and measure architectural debt, by analyzing a wide range of ATD symptoms which to date have only marginally studied, in order to achieve a comprehensive, empirically supported, and straightforward ATD management strategy.

In order to reach our goal, we exploit a vast range of scientific methods to have further insights into the ATD phenomenon, and create data-driven approaches to enrich our ATD body of knowledge, identifying which information is most useful to detect ATD, and how this data can be leveraged to support developers. Specifically, one of our research pillars deals with characterizing which previously unexplored, quantifiable, symptoms point to the presence of ATD in software-intensive systems. This is achieved by rooting our research in continuous and ongoing collaborations with industrial parties, investigating from an academic viewpoint how ATD manifests itself in real-world contexts.

Additionally, we leverage existing findings and approaches in order to take full advantage of the existing tools and techniques to support, enrich, and refine our findings. This enables us to consider and analyze a heterogeneous set of software artifacts in which ATD can manifests itself, leading to a comprehensive coverage of the known dimensions of ATD.

Our ultimate goal is to conceive an approach providing a lightweight, communicable, and actionable overview of the ATD present in software-intensive systems. On one hand, an ATD overview can provide and intuitive visualization of ATD, supporting conversations, understanding, and communication of the current state of ATD in software-intensive systems. On the other hand, by enhancing the overview thanks to the different levels of granularity of the derived data utilized to build the overview, it is possible to provide deep, actionable, data-driven guidance on where the architectural debt is rooted, enabling developers to pinpoint the location of ATD in large and complex architectures, providing them with the necessary tool to manage ATD before it is too late. Our research standpoint is that, in the near future, ATD will not be anymore an invisible beast which taunts developers during their “coffee machine talks”, but rather a domesticated animal of which everybody knows the presence, and everybody goes along with.

Bibliography

[1] R. Verdecchia, I. Malavolta, and P. Lago, "Architectural technical debt identification: The research landscape," in *IEEE/ACM International Conference on Technical Debt (TechDebt)*, 2018, pp. 11–20.

[2] I. Ozkaya, "Managing technical debt," 2017, international Software Architecture PhD School (iSAPS), unpublished.

[3] W. Cunningham, "The wycash portfolio management system," *OOPS Messenger*, vol. 4, no. 2, pp. 29–30, 1993.

[4] N. S. Alves, T. S. Mendes, M. G. de Mendonça, R. O. Spínola, F. Shull, and C. Seaman, "Identification and management of technical debt: A systematic mapping study," *Information and Software Technology*, vol. 70, pp. 100–121, Feb. 2016.

[5] A. Ampatzoglou, A. Ampatzoglou, A. Chatzigeorgiou, and P. Avgeriou, "The financial aspect of managing technical debt: A systematic literature review," *Information and Software Technology*, vol. 64, pp. 52–73, 2015.

[6] M. Flower, "TechnicalDebtQuadrant," 2009, (Accessed 22nd September 2017). [Online]. Available: https://martinfowler.com/bliki/TechnicalDebtQuadrant.html

[7] S. McConnell, "Technical Debt," Accessed 28th September 2017. [Online]. Available: http://www.construx.com/10x_Software_Development/Technical_Debt/

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

[9] Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management," *Journal of Systems and Software*, vol. 101, no. 3, pp. 193–220, 2015.

[10] E. Tom, A. Aurum, and R. Vidgen, "An exploration of technical debt," *Journal of Systems and Software*, vol. 86, no. 6, pp. 1498–1516, Jun. 2013.

[11] T. Besker, A. Martini, and J. Bosch, "A Systematic Literature Review and a Unified Model of ATD." IEEE, Aug. 2016, pp. 189–197.

[12] C. Izurieta, A. Vetrò, N. Zazworka, Y. Cai, C. Seaman, and F. Shull, "Organizing the technical debt landscape," in *Proceedings of the Third International Workshop on Managing Technical Debt*. IEEE Press, 2012, pp. 23–26.

[13] C. Izurieta, I. Ozkaya, C. Seaman, and W. Snipes, "Technical Debt: A Research Roadmap Report on the Eighth Workshop on Managing Technical Debt (MTD 2016)," *ACM SIGSOFT Software Engineering Notes*, vol. 42, no. 1, pp. 28–31, 2017. [Online]. Available: http://dl.acm.org/citation.cfm?id=3041774

[14] D. Falessi, Z. Codabux, G. Rong, I. Stamelos, W. Ferreira, I. S. Wiese, E. Barreiros, C. Quesada-Lopez, and P. Tsirakidis, "Trends in empirical research: the report on the 2014 Doctoral Symposium on Empirical Software Engineering," *ACM SIGSOFT Software Engineering Notes*, vol. 40, no. 5, pp. 30–35, Sep. 2015. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2815021.2815034

[15] "TechDebt 2018 International Conference on Technical Debt - TechDebt 2018," accessed 26th September 2017. [Online]. Available: https://2018.techdebtconf.org/track/TechDebt-2018-papers

[16] N. A. Ernst, "On the role of requirements in understanding and managing technical debt," in *Proceedings of the Third International Workshop on Managing Technical Debt*. IEEE Press, 2012, pp. 61–64.

[17] I. Ozkaya, R. L. Nord, H. Koziolek, and P. Avgeriou, "Toward Simpler, not Simplistic, Quantification of Software Architecture and Metrics: Report on the Second International Workshop on Software Architecture and Metrics," *ACM SIGSOFT Software Engineering Notes*, vol. 40, no. 5, pp. 43–46, Sep. 2015. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2815021.2815037

[18] R. Kazman, Y. Cai, R. Mo, Q. Feng, L. Xiao, S. Haziyev, V. Fedak, and A. Shapochka, "A Case Study in Locating the Architectural Roots of Technical Debt." IEEE, May 2015, pp. 179–188.

[19] C. Fernandez-Sanchez, J. Garbajosa, C. Vidal, and A. Yague, "An Analysis of Techniques and Methods for Technical Debt Management: A Reflection from the Architecture Perspective." IEEE, May 2015, pp. 22–28. [Online]. Available: http://ieeexplore.ieee.org/document/7174845/

[20] Z. Li, P. Liang, and P. Avgeriou, "Architectural Technical Debt Identification Based on Architecture Decisions and Change Scenarios." IEEE, May 2015, pp. 65–74. [Online]. Available: http://ieeexplore.ieee.org/document/7158505/

[21] A. Martini, E. Sikander, and N. Madlani, "A Semi-Automated Framework for the Identification and Estimation of Architectural Technical Debt: A comparative case-study on the modularization of a software component," *Information and Software Technology*, Sep. 2017. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S095058491630355X

[22] R. L. Nord, I. Ozkaya, P. Kruchten, and M. Gonzalez-Rojas, "In Search of a Metric for Managing Architectural Technical Debt." IEEE, Aug. 2012, pp. 91–100. [Online]. Available: http://ieeexplore.ieee.org/document/6337765/

[23] A. Martini, J. Bosch, and M. Chaudron, "Investigating Architectural Technical Debt accumulation and refactoring over time: A multiple-case study," *Information and Software Technology*, vol. 67, pp. 237–253, Nov. 2015. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0950584915001287

[24] A. Martini and J. Bosch, "Towards Prioritizing Architecture Technical Debt: Information Needs of Architects and Product Owners." IEEE, Aug. 2015, pp. 422–429. [Online]. Available: http://ieeexplore.ieee.org/document/7302484/

[25] N. Zazworka, R. O. Spínola, A. Vetro, F. Shull, and C. Seaman, "A case study on effectively identifying technical debt," in *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*. ACM, 2013, pp. 42–47.

[26] W. H. Brown, R. C. Malveau, H. W. S. McCormick, and T. J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*, 1st ed. New York, NY, USA: John Wiley & Sons, Inc., 1998.

[27] R. J. Eisenberg, "A threshold based approach to technical debt," *ACM SIGSOFT Software Engineering Notes*, vol. 37, no. 2, pp. 1–6, 2012.

[28] I. Gat and J. D. Heintz, "From assessment to reduction: how cutter consortium helps rein in millions of dollars in technical debt," in *Proceedings of the 2nd Workshop on Managing Technical Debt*. ACM, 2011, pp. 24–26.

[29] J. D. Morgenthaler, M. Gridnev, R. Sauciuc, and S. Bhansali, "Searching for build debt: Experiences managing technical debt at google," in *Proceedings of the Third International Workshop on Managing Technical Debt*. IEEE Press, 2012, pp. 1–6.

[30] S. Stolberg, "Enabling agile testing through continuous integration," in *Agile Conference, 2009. AGILE'09.* IEEE, 2009, pp. 369–374.

[31] T. Coq and J.-P. Rosen, "The sqale quality and analysis models for assessing the quality of ada source code," in *International Conference on Reliable Software Technologies*. Springer, 2011, pp. 61–74.

[32] J. Heintz, "Modernizing the delorean system: Comparing actual and predicted results of a technical debt reduction project," *Cutter IT Journal*, vol. 23, no. 10, p. 7, 2010.

[33] D. Falessi, M. A. Shaw, F. Shull, K. Mullen, and M. S. Keymind, "Practical considerations, challenges, and requirements of tool-support for managing technical debt," in *Managing Technical Debt (MTD), 2013 4th International Workshop on*. IEEE, 2013, pp. 16–19.

[34] S. Akbarinasaji, "Toward measuring defect debt and developing a recommender system for their prioritization," in *Proceedings of the 13th International Doctoral Symposium on Empirical Software Engineering*, 2015, pp. 1–6.

[35] N. S. Alves, L. F. Ribeiro, V. Caires, T. S. Mendes, and R. O. Spínola, "Towards an ontology of terms on technical debt," in *Managing Technical Debt (MTD), 2014 Sixth International Workshop on*. IEEE, 2014, pp. 1–7.

[36] D. A. Tamburri, P. Kruchten, P. Lago, and H. Van Vliet, "Social debt in software engineering: insights from industry," *Journal of Internet Services and Applications*, vol. 6, no. 1, p. 10, 2015.

[37] K. Power, "Understanding the impact of technical debt on the capacity and velocity of teams and organizations: Viewing team and organization capacity as a portfolio of real options," in *Managing Technical Debt (MTD), 2013 4th International Workshop on*. IEEE, 2013, pp. 28–31.

[38] W. Cunningham, "The WyCash portfolio management system," *ACM SIGPLAN OOPS Messenger*, vol. 4, no. 2, pp. 29–30, 1993. [Online]. Available: http://dl.acm.org/citation.cfm?id=157715

[39] J. S. van der Ven and J. Bosch, "Making the right decision: supporting architects with design decision data," in *European Conference on Software Architecture*. Springer, 2013, pp. 176–183. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-642-39031-9_15

[40] A. Martini, J. Bosch, and M. Chaudron, "Architecture Technical Debt: Understanding Causes and a Qualitative Model." IEEE, Aug. 2014, pp. 85–92. [Online]. Available: http://ieeexplore.ieee.org/document/6928795/

[41] H. van Vliet, *Software engineering: principles and practice*, 3rd ed. Wiley, 1993.

[42] Z. Li, P. Liang, and P. Avgeriou, "Architectural debt management in value-oriented architecting," *Economics-Driven Software Architecture, Elsevier*, pp. 183–204, 2014.

[43] Z. Li, P. Liang, P. Avgeriou, N. Guelfi, and A. Ampatzoglou, "An empirical investigation of modularity metrics for indicating architectural technical debt," in *Proceedings of the 10th international ACM Sigsoft conference on Quality of software architectures*. ACM, 2014, pp. 119–128.

[44] N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, and I. Gorton, "Measure it? manage it? ignore it? software practitioners and technical debt," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 50–60.

[45] A. Martini and J. Bosch, "The danger of architectural technical debt: Contagious debt and vicious circles," in *2015 12th Working IEEE/IFIP Conference on Software Architecture*. IEEE, 2015, pp. 1–10.

[46] B. Curtis, J. Sappidi, and A. Szynkarski, "Estimating the size, cost, and types of technical debt," in *Proceedings of the Third International Workshop on Managing Technical Debt*. IEEE Press, 2012, pp. 49–53.

[47] F. A. Fontana, V. Ferme, and M. Zanoni, "Towards assessing software architecture quality by exploiting code smell relations," in *Proceedings of the Second International Workshop on Software Architecture and Metrics*. IEEE Press, 2015, pp. 1–7.

[48] E. Ligu, A. Chatzigeorgiou, T. Chaikalis, and N. Ygeionomakis, "Identification of refused bequest code smells," in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. IEEE, 2013, pp. 392–395.

[49] B. Boehm, "Architecture-based quality attribute synergies and conflicts," in *Software Architecture and Metrics (SAM), 2015 IEEE/ACM 2nd International Workshop on*. IEEE, 2015, pp. 29–34.

[50] R. Verdecchia, I. Malavolta, and P. Lago, "Architectural Technical Debt Identification: The Research Landscape," in *International Conference on Technical Debt (TechDebt)*, 2018.

[51] I. Malavolta, R. Verdecchia, M. Bruntink, B. Filipovic, and P. Lago, "How Maintainability Issues of Android Apps Evolve," in *International Conference on Software Maintenance and Evolution*, 2018.

[52] R. Verdecchia, "Identifying Architectural Technical Debt in Android Applications through Compliance Checking," in *International Conference on Mobile Software Engineering and Systems*, 2018.

[53] R. Verdecchia, I. Malavolta, and P. Lago, "Guidelines for architecting android apps: A mixed-method empirical study," in *2019 IEEE International Conference on Software Architecture*. IEEE, 2019, pp. 141–150.

[54] R. Verdecchia, P. Lago, I. Malavolta, and I. Ozkaya, "ATDx: Building an architectural technical debt index." in *ENASE*, 2020, pp. 531–539.

[55] ——, "Empirical evaluation of an architectural technical debt index in the context of the apache and onap ecosystems," in *Journal article under submission*, 2021.

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

[57] R. Verdecchia, P. Kruchten, P. Lago, and I. Malavolta, "Building and evaluating a theory of architectural technical debt in software-intensive systems," *Journal of Systems and Software*, 2021.

[58] S. Ospina, R. Verdecchia, I. Malavolta, and P. Lago, "ATDx: A tool for providing a data-driven overview of architectural technical debt in software-intensive systems," *European Conference on Software Architecture (to appear)*, 2021.

[59] J. Bogner, R. Verdecchia, and I. Gerostathopoulos, "Characterizing technical debt and antipatterns in ai-based systems: A systematic mapping study," *International Conference on Technical Debt*, 2021.

[60] R. Verdecchia, P. Lago, I. Malavolta, and I. Ozkaya, "ATDx: prototype implementation technical report," in *VU Technical Reports*, 2020.

[61] R. Verdecchia, A. Guldner, Y. Becker, and E. Kern, "Code-level energy hotspot localization via naive spectrum based testing," in *Advances and New Trends in Environmental Informatics - Managing Disruption, Big Data and Open Science*, 2018, pp. 111–130. [Online]. Available: https://doi.org/10.1007/978-3-319-99654-7_8

[62] R. Verdecchia, R. A. Saez, G. Procaccianti, and P. Lago, "Empirical evaluation of the energy impact of refactoring code smells," in *5th International Conference on Information and Communication Technology for Sustainability, ICT4S 2018, Toronto, Canada, May 14-18, 2018*, 2018, pp. 365–383. [Online]. Available: http://www.easychair.org/publications/paper/MxpT

[63] A. Bener, B. Turhan, and S. Biffl, Eds., *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2017, Toronto, ON, Canada, November 9-10, 2017*. IEEE Computer Society, 2017. [Online]. Available: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=8169617

[64] P. Lago, R. Verdecchia, N. Condori-Fernandez, E. Rahmadian, J. Sturm, T. van Nijnanten, R. Bosma, C. Debuysscher, and P. Ricardo, "Designing for sustainability: Lessons learned from four industrial projects," in *Advances and New Trends in Environmental Informatics*. Springer, 2021, pp. 3–18.

[65] R. Verdecchia, P. Lago, and C. de Vries, "The leap technology roadmap: Lower energy acceleration program (leap) solutions, adoption factors, impediments, open problems, and scenarios," *VU Technical Reports*, 2021.

[66] R. Verdecchia, E. Cruciani, B. Miranda, and A. Bertolino, "Know you neighbor: Fast static prediction of test flakiness," *IEEE Access*, 2021.

[67] F. Corò, R. Verdecchia, E. Cruciani, B. Miranda, and A. Bertolino, "JTeC: A Large Collection of Java Test Classes for Test Code Analysis and Processing," in *Under Submission, year = 2020*.

[68] E. Cruciani, B. Miranda, R. Verdecchia, and A. Bertolino, "Scalable approaches for test suite reduction," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, 2019, pp. 419–429. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00055

[69] B. Miranda, E. Cruciani, R. Verdecchia, and A. Bertolino, "FAST approaches to scalable similarity-based test case prioritization," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, 2018, pp. 222–232. [Online]. Available: https://doi.org/10.1145/3180155.3180210

[70] P. Lago, J. F. Cai, R. C. de Boer, P. Kruchten, and R. Verdecchia, "Decidarch: Playing cards as software architects," in *52nd Hawaii International Conference on System Sciences, HICSS 2019, Grand Wailea, Maui, Hawaii, USA, January 8-11, 2019*, 2019, pp. 1–10. [Online]. Available: http://hdl.handle.net/10125/60220

[71] R. C. de Boer, P. Lago, R. Verdecchia, and P. Kruchten, "Decidarch V2: an improved game to teach architecture design decision making," in *IEEE International Conference on Software Architecture Companion, ICSA Companion 2019, Hamburg, Germany, March 25-26, 2019*, 2019, pp. 153–157. [Online]. Available: https://doi.org/10.1109/ICSA-C.2019.00034

[72] T. Besker, A. Martini, and J. Bosch, "Managing architectural technical debt: A unified model and systematic literature review," *Journal of Systems and Software*, vol. 135, pp. 1–16, 2018.

[73] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[74] B. Kitchenham and P. Brereton, "A systematic review of systematic review process research in software engineering," *Information and software technology*, vol. 55, no. 12, pp. 2049–2075, 2013.

[75] V. R. Basili, G. Caldiera, and H. D. Rombach, "The Goal Question Metric Approach," in *Encyclopedia of Software Engineering*. Wiley, 1994, vol. 2, pp. 528–532.

[76] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '14. New York, NY, USA: ACM, 2014, pp. 38:1–38:10.

[77] K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson, "Systematic mapping studies in software engineering," in *Proceedings of the 12th International Conference*

on Evaluation and Assessment in Software Engineering, ser. EASE'08. British Computer Society, 2008, pp. 68–77.

[78] T. Greenhalgh and R. Peacock, "Effectiveness and efficiency of search methods in systematic reviews of complex evidence: audit of primary sources," *BMJ*, vol. 331, no. 7524, pp. 1064–1065, 2005.

[79] "Scopus | The largest database of peer-reviewed literature | Elsevier," 2018, (Accessed 22nd January 2018). [Online]. Available: https://www.elsevier.com/solutions/scopus

[80] K. Petersen, S. Vakkalanka, and L. Kuzniarz, "Guidelines for conducting systematic mapping studies in software engineering: An update," *Information and Software Technology*, vol. 64, pp. 1–18, 2015.

[81] M. Ivarsson and T. Gorschek, "A method for evaluating rigor and industrial relevance of technology evaluations," *Empirical Software Engineering*, vol. 16, no. 3, pp. 365–395, 2011.

[82] B. A. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," Keele University and University of Durham, Tech. Rep., 2007.

[83] P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman, "Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162)," in *Dagstuhl Reports*, vol. 6, no. 4. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[84] C. Y. Baldwin and K. B. Clark, *Design rules: The power of modularity*. MIT press, 2000, vol. 1.

[85] J. Knodel, M. Lindvall, D. Muthig, and M. Naab, "Static evaluation of software architectures," in *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*. IEEE, 2006, pp. 10–pp.

[86] Z. Li, P. Liang, and P. Avgeriou, "Chapter 9 - architectural debt management in value-oriented architecting," in *Economics-Driven Software Architecture*, I. Mistrik, , R. Bahsoon, , R. Kazman, , and Y. Zhang, Eds. Boston: Morgan Kaufmann, 2014, pp. 183 – 204.

[87] N. Alves, T. S. Mendes, M. G. de Mendonça, R. O. Spínola, F. Shull, and C. Seaman, "Identification and management of technical debt: A systematic mapping study," *Information and Software Technology*, vol. 70, pp. 100–121, 2016.

[88] A. Potdar and E. Shihab, "An exploratory study on self-admitted technical debt," in *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE, 2014, pp. 91–100.

[89] B. Selic, "Agile documentation, anyone?" *IEEE software*, vol. 26, no. 6, 2009.

[90] L. Xiao, Y. Cai, and R. Kazman, "Titan: A toolset that connects software architecture with quality analysis," in *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 763–766.

[91] G. Samarthyam, M. Muralidharan, and R. K. Anna, "Understanding test debt," *Trends in Software Testing*, pp. 1–17, 2017.

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

[93] L. Xiao, Y. Cai, R. Kazman, R. Mo, and Q. Feng, "Identifying and quantifying architectural debt," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 488–498.

[94] F. Arcelli Fontana, I. Pigazzini, R. Roveda, D. Tamburri, M. Zanoni, and E. Di Nitto, "Arcan: A tool for architectural smells detection," in *IEEE International Conference on Software Architecture Workshops (ICSAW)*. IEEE, 2017, pp. 282–285.

[95] A. Martini, E. Sikander, and N. Madlani, "A semi-automated framework for the identification and estimation of architectural technical debt: A comparative case-study on the modularization of a software component," *Information and Software Technology*, vol. 93, pp. 264–279, 2018.

[96] T. Kuipers and J. Visser, "A tool-based methodology for software portfolio monitoring." in *Software Audit and Metrics*, 2004, pp. 118–128.

[97] M. Keeling, *Design It!* Pragmatic Bookshelf, Oct. 2017. [Online]. Available: https://www.oreilly.com/library/view/design-it/9781680502923/

[98] H. Wang and M. Song, "Ckmeans.1d.dp: Optimal k-means clustering in one dimension by dynamic programming," *The R journal*, vol. 3, no. 2, p. 29, 2011.

[99] W. Lidwell, K. Holden, and J. Butler, *Universal Principles of Design*. Rockport Pub, 2010.

[100] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "An In-depth Study of the Promises and Perils of Mining GitHub," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2035–2071, 2016.

[101] G. Caldiera, V. R. Basili, and H. D. Rombach, "Goal question metric paradigm," *Encyclopedia of software engineering*, vol. 1, pp. 528–532, 1994.

[102] A. E. Hassan, "The road ahead for mining software repositories," in *2008 Frontiers of Software Maintenance*. IEEE, 2008, pp. 48–57.

[103] N. Fenton and J. Bieman, *Software metrics: a rigorous and practical approach*. CRC press, 2014, pp. 33–40.

[104] E. Smith, R. Loftin, E. Murphy-Hill, C. Bird, and T. Zimmermann, "Improving developer participation rates in surveys," in *2013 6th International workshop on cooperative and human aspects of software engineering (CHASE)*. IEEE, 2013, pp. 89–92.

[105] A. Janes, V. Lenarduzzi, and A. C. Stan, "A continuous software quality monitoring approach for small and medium enterprises," in *8th ACM/SPEC on International Conference on Performance Engineering Companion*, 2017, pp. 97–100.

[106] N. A. Ernst, S. Bellomo, I. Ozkaya, and R. L. Nord, "What to fix? distinguishing between design and non-design rules in automated tools," in *IEEE International Conference on Software Architecture (ICSA)*, 2017, pp. 165–168.

[107] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, no. 4, pp. 308–320, 1976.

[108] K. Manikas and K. M. Hansen, "Software ecosystems–a systematic literature review," *Journal of Systems and Software*, vol. 86, no. 5, pp. 1294–1306, 2013.

[109] I. Malavolta, R. Verdecchia, B. Filipovic, M. Bruntink, and P. Lago, "How Maintainability Issues of Android Apps Evolve," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 334–344.

[110] I. B. Weiner and W. E. Craighead, *The Corsini Encyclopedia of Psychology, Volume 4*. John Wiley & Sons, 2010, vol. 2, pp. 637–638.

[111] I. Malavolta, G. Lewis, B. Schmerl, P. Lago, and D. Garlan, "How do you architect your robots? state of the practice and guidelines for ros-based systems," in *2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2020, pp. 31–40.

[112] G. Digkas, M. Lungu, A. Chatzigeorgiou, and P. Avgeriou, "The evolution of technical debt in the apache ecosystem," in *European Conference on Software Architecture*. Springer, 2017, pp. 51–66.

[113] V. Lenarduzzi, N. Saarimaki, and D. Taibi, "On the diffuseness of code technical debt in java projects of the apache ecosystem," in *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*. IEEE, 2019, pp. 98–107.

[114] J. Tan, M. Lungu, and P. Avgeriou, "Towards studying the evolution of technical debt in the python projects from the apache software ecosystem." in *BENEVOL*, 2018, pp. 43–45.

[115] Z. Li, Q. Yu, P. Liang, R. Mo, and C. Yang, "Interest of defect technical debt: An exploratory study on apache projects," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 629–639.

[116] F. Arcelli Fontana, R. Roveda, and M. Zanoni, "Tool support for evaluating architectural debt of an existing system: An experience report," in *Annual ACM Symposium on Applied Computing*, 2016, pp. 1347–1349.

[117] A. Martini, F. A. Fontana, A. Biaggi, and R. Roveda, "Identifying and prioritizing architectural debt through architectural smells: a case study in a large software company," in *European Conference on Software Architecture*. Springer, 2018, pp. 320–335.

[118] R. Roveda, F. Arcelli Fontana, I. Pigazzini, and M. Zanoni, "Towards an architectural debt index," in *44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 2018, pp. 408–416.

[119] R. Kazman, Y. Cai, R. Mo, Q. Feng, L. Xiao, S. Haziyev, V. Fedak, and A. Shapochka, "A case study in locating the architectural roots of technical debt," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2. IEEE, 2015, pp. 179–188.

[120] Y. Cai and R. Kazman, "Detecting and quantifying architectural debt: theory and practice," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion*. IEEE, 2017, pp. 503–504.

[121] ——, "Dv8: automated architecture analysis tool suites," in *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*. IEEE, 2019, pp. 53–54.

[122] R. L. Nord, I. Ozkaya, P. Kruchten, and M. Gonzalez-Rojas, "In search of a metric for managing architectural technical debt," in *Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, 2012, pp. 91–100.

[123] D. M. Le, D. Link, A. Shahbazian, and N. Medvidovic, "An empirical study of architectural decay in open-source software," in *IEEE International Conference on Software Architecture (ICSA)*, 2018, pp. 176–185.

[124] J. W. Tukey, "Exploratory data analysis addision-wesley," *Reading, MA*, vol. 688, 1977.

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

[126] 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, p. 106377, 2020.

[127] T. L. Alves, C. Ypma, and J. Visser, "Deriving metric thresholds from benchmark data," in *2010 IEEE International Conference on Software Maintenance*. IEEE, 2010, pp. 1–10.

[128] M. Ulan, W. Löwe, M. Ericsson, and A. Wingkvist, "Towards meaningful software metrics aggregation," in *Proceedings of the 18th Belgium-Netherlands Software Evolution Workshop*, 2019.

[129] W. Cunningham, "The wycash portfolio management system," in *OOPSLA'92 proceedings*, 1992, Conference Proceedings.

[130] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, R. Sangwan, C. B. Seaman, K. Sullivan, and N. Zazworka, "Managing technical debt in software-intensive systems," in *Future of Software Engineering Research (FoSER 2010) Workshop, part of FSE 2010*. ACM, 2010, Conference Proceedings, pp. 47–52.

[131] P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman, *Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162)*. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, vol. 6.

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

[133] R. Verdecchia, P. Kruchten, and P. Lago, "Architectural Technical Debt: A Grounded Theory," in *European Conference on Software Architecture*. Springer, 2020, pp. 202–219.

[134] R. S. Schreiber, P. N. Stern *et al.*, *Using grounded theory in nursing*. Springer Publishing Company, 2001.

[135] M. Kenny and R. Fourie, "Contrasting classic, straussian, and constructivist grounded theory: methodological and philosophical conflicts," *The Qualitative Report*, vol. 20, no. 8, pp. 1270–1289, 2015.

[136] B. G. Glaser and A. L. Strauss, *Discovery of grounded theory: Strategies for qualitative research*. Aldine, 1967.

[137] B. G. Glaser, *The Grounded Theory Perspective III: Theoretical Coding*. Sociology Press, 2005.

[138] B. Glaser, "Theoretical sensitivity," *Advances in the methodology of grounded theory*, 1978.

[139] ——, *Theoretical Sensitivity*. Sociology Press, 1978.

[140] B. G. Glaser, *Basics of grounded theory analysis: Emergence vs forcing*. Sociology press, 1992.

[141] K.-J. Stol, P. Ralph, and B. Fitzgerald, "Grounded theory in software engineering research: a critical review and guidelines," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 120–131.

[142] S. Adolph, W. Hall, and P. Kruchten, "Using grounded theory to study the experience of software development," *Empirical Software Engineering*, vol. 16, no. 4, pp. 487–513, 2011.

[143] H. J. Rubin and I. S. Rubin, *Qualitative interviewing: The art of hearing data*. Sage Publications, 2011.

[144] K. Rose, "Unstructured and semi-structured interviewing." *Nurse Researcher*, vol. 1, no. 3, pp. 23–32, 1994.

[145] J. M. Morse, ""Emerging From the Data": The Cognitive Processes of Analysis in Qualitative Inquiry," *Critical issues in qualitative research methods*, pp. 23–46, 1994.

[146] D. G. Oliver, J. M. Serovich, and T. L. Mason, "Constraints and opportunities with interview transcription: Towards reflection in qualitative research," *Social forces*, vol. 84, no. 2, pp. 1273–1289, 2005.

[147] R. Hoda and J. Noble, "Becoming agile: a grounded theory of agile transitions in practice," in *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 2017, pp. 141–151.

[148] H. Engward, "Understanding grounded theory," *Nursing standard*, vol. 28, no. 7, 2013.

[149] J. Kontio, J. Bragge, and L. Lehtola, "The focus group method as an empirical tool in software engineering," in *Guide to advanced empirical software engineering*. Springer, 2008, pp. 93–116.

[150] K. Lomborg and M. Kirkevold, "Truth and validity in grounded theory–a reconsidered realist interpretation of the criteria: fit, work, relevance and modifiability," *Nursing Philosophy*, vol. 4, no. 3, pp. 189–200, 2003.

[151] A. Bryman, *Social research methods*. Oxford University Press, 2001.

[152] ISO/IEC/IEEE, "Systems and software engineering – architecture description," *ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000)*, pp. 1–46, 1 2011.

[153] A. Strauss and J. Corbin, *Basics of qualitative research techniques*. Sage publications Thousand Oaks, 1998.

[154] P. Kruchten, "What Colour Is Your Backlog?" 2008, available Online: https://tinyurl.com/y6f7vhpx (Accessed 29th September 2019).

[155] A. Zerouali, E. Constantinou, T. Mens, G. Robles, and J. González-Barahona, "An empirical analysis of technical lag in npm package dependencies," in *International Conference on Software Reuse*. Springer, 2018, pp. 95–110.

[156] A. A. Almonaies, J. R. Cordy, and T. R. Dean, "Legacy system evolution towards service-oriented architecture," in *International Workshop on SOA Migration and Evolution*, 2010, pp. 53–62.

[157] J. Kruger and D. Dunning, "Unskilled and unaware of it: how difficulties in recognizing one's own incompetence lead to inflated self-assessments." *Journal of personality and social psychology*, vol. 77, no. 6, p. 1121, 1999.

[158] M. E. Conway, "How do committees invent," *Datamation*, vol. 14, no. 4, pp. 28–31, 1968.

[159] A. Wert, M. Oehler, C. Heger, and R. Farahbod, "Automatic detection of performance anti-patterns in inter-component communications," in *Proceedings of the 10th international ACM Sigsoft conference on Quality of software architectures*. ACM, 2014, pp. 3–12.

[160] C. Mateos, A. Zunino, A. Flores, and S. Misra, "Cobol systems migration to SOA: assessing antipatterns and complexity," *Information Technology and Control*, vol. 48, no. 1, pp. 71–89, 2019.

[161] C.-C. Chiang and C. Bayrak, "Legacy software modernization," in *2006 IEEE international conference on systems, man and cybernetics*, vol. 2. IEEE, 2006, pp. 1304–1309.

[162] D. Myers and S. M. Smith, *Exploring social psychology*. McGraw-Hill Ryerson, 2015.

[163] D. Kahneman and A. Tversky, "Intuitive prediction: Biases and corrective procedures," Cambridge University Press, Tech. Rep., 1977.

[164] A. Martini and J. Bosch, "On the interest of architectural technical debt: uncovering the contagious debt phenomenon," *Journal of Software: Evolution and Process*, 2017.

[165] A. Martini, J. Bosch, and M. Chaudron, "Investigating Architectural Technical Debt accumulation and refactoring over time: A multiple-case study," *Information and Software Technology*, vol. 67, pp. 237–253, 2015.

[166] Z. Li, P. Liang, and P. Avgeriou, "Architecture viewpoints for documenting architectural technical debt," in *Software Quality Assurance*. Elsevier, 2016, pp. 85–132.

[167] C. Urquhart, H. Lehmann, and M. D. Myers, "Putting the 'theory' back into grounded theory: guidelines for grounded theory studies in information systems," *Information systems journal*, vol. 20, no. 4, pp. 357–381, 2010.

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

[169] F. P. Brooks Jr, "The mythical man-month (anniversary edition). Addison Wesley, Boston," 1995.

[170] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[171] C. Szyperski, D. Gruntz, and S. Murer, *Component software: beyond object-oriented programming*. Pearson Education, 2002.

[172] A. Martini and J. Bosch, "Towards prioritizing architecture technical debt: information needs of architects and product owners," in *2015 41st Euromicro Conference on Software Engineering and Advanced Applications*. IEEE, 2015, pp. 422–429.

[173] H. Ghanbari, T. Besker, A. Martini, and J. Bosch, "Looking for peace of mind?: manage your (technical) debt: an exploratory field study," in *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE Press, 2017, pp. 384–393.

[174] B. Martin, "Global Digital Future in Focus - 2018 International Edition," 2018, comscore White Paper.

[175] "Android developer portal," 2018. [Online]. Available: http://developer.android.com/about/index.html

[176] P. Bourque and R. E. Fairley, *Guide to the Software Engineering Body of Knowledge, Version 3.0*. IEEE Computer Society, 2014.

[177] "Facebook app release history," 2017. [Online]. Available: http://www.apk4fun.com/history/2430

[178] "Android core app quality guidelines," 2017. [Online]. Available: http://developer.android.com/develop/quality-guidelines/core-app-quality.html

[179] J. Visser, "SIG/TÜViT evaluation criteria trusted product maintainability: Guidance for producers," Software Improvement Group, 9.0, February 2017. [Online]. Available: https://www.sig.eu/wp-content/uploads/2016/10/SIG-TUViT-Evaluation-Criteria-Trusted-Product-Maintainability-Guidance-for-producers.pdf

[180] "Sig official website," 2017. [Online]. Available: http://sig.eu

[181] M. D. Penta, L. Cerulo, and L. Aversano, "The life and death of statically detected vulnerabilities: An empirical study," *Information and Software Technology*, vol. 51, no. 10, pp. 1469 – 1484, 2009.

[182] I. 25010:2011, "Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models," 2011. [Online]. Available: https://www.iso.org/standard/35733.html

[183] "Android lint official website," 2017. [Online]. Available: http://developer.android.com/studio/write/lint.html

[184] "Findbugs official website," 2017. [Online]. Available: http://findbugs.sourceforge.net

[185] "Sonarqube," 2018. [Online]. Available: https://www.sonarqube.org/

[186] "Pmd official website," 2017. [Online]. Available: http://pmd.github.io

[187] "Checkstyle official website," 2017. [Online]. Available: checkstyle.sourceforge.net

[188] "Jarchitect," 2018. [Online]. Available: https://www.jarchitect.com

[189] "Scitools," 2018. [Online]. Available: https://scitools.com

[190] R. Baggen, J. P. Correia, K. Schill, and J. Visser, "Standardized code quality benchmarking for improving software maintainability," *Software Quality Journal*, vol. 20, no. 2, pp. 287–307, 2012.

[191] D. Bijlsma, M. A. Ferreira, B. Luijten, and J. Visser, "Faster issue resolution with higher technical quality of software," *Software Quality Journal*, vol. 20, no. 2, pp. 265–285, Jun. 2012.

[192] T. Döhmen, M. Bruntink, D. Ceolin, and J. Visser, "Towards a Benchmark for the Maintainability Evolution of Industrial Software Systems," in *IWSM-MENSURA 2016 Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement*. IEEE, 2016, pp. 11–22.

[193] T. Das, M. D. Penta, and I. Malavolta, "A Quantitative and Qualitative Investigation of Performance-Related Commits in Android Apps," in *ICSME '16 Proceedings of the 32nd International Conference on Software Maintenance and Evolution*. IEEE, 2016, pp. 443–448.

[194] "Fdroid," 2017. [Online]. Available: http://f-droid.org

[195] "Wikipedia page on open-source android apps," 2017. [Online]. Available: http://en.wikipedia.org/wiki/List_of_free_and_open-source_Android_applications

[196] "How long does it take to build a mobile app?" 2017. [Online]. Available: http://www.kinvey.com/how-long-to-build-an-app-infographic/

[197] G. Hecht, O. Benomar, R. Rouvoy, N. Moha, and L. Duchien, "Tracking the Software Quality of Android Applications along their Evolution," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015, pp. 429–436.

[198] T. Döhmen, M. Bruntink, D. Ceolin, and J. Visser, "Towards a benchmark for the maintainability evolution of industrial software systems," in *Software Measurement and the International Conference on Software Process and Product Measurement (IWSM-MENSURA), 2016 Joint Conference of the International Workshop on*. IEEE, 2016, pp. 11–21.

[199] D. Bijlsma, M. A. Ferreira, B. Luijten, and J. Visser, "Faster issue resolution with higher technical quality of software," *Software quality journal*, vol. 20, no. 2, pp. 265–285, 2012.

[200] I. Heitlager, T. Kuipers, and J. Visser, "A practical model for measuring maintainability," in *Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the*. IEEE, 2007, pp. 30–39.

[201] R. Koschke, R. Falke, and P. Frenzel, "Clone detection using abstract syntax suffix trees," in *Reverse Engineering, 2006. WCRE'06. 13th Working Conference on*. IEEE, 2006, pp. 253–262.

[202] J. Rosenberg, "Statistical methods and measurement," in *Guide to Advanced Empirical Software Engineering*. Springer, 2008, pp. 155–184.

[203] P. J. Brockwell and R. A. Davis, *Time series: theory and methods*. Springer Science & Business Media, 2013.

[204] A. I. McLeod, H. Yu, and E. Mahdi, "Time series analysis with r," in *Handbook of statistics*. Elsevier, 2012, vol. 30, pp. 661–712.

[205] D. A. Dickey and W. A. Fuller, "Distribution of the estimators for autoregressive time series with a unit root," *Journal of the American statistical association*, vol. 74, no. 366a, pp. 427–431, 1979.

[206] B. L. Bowerman and R. T. O'Connell, *Forecasting and time series: An applied approach. 3rd Edition*. Duxbury press, 1993.

[207] R. B. Cleveland, W. S. Cleveland, and I. Terpenning, "Stl: A seasonal-trend decomposition procedure based on loess," *Journal of Official Statistics*, vol. 6, no. 1, p. 3, 1990.

[208] A. Atchison, C. Berardi, N. Best, E. Stevens, and E. Linstead, "A time series analysis of travistorrent builds: to everything there is a season," in *Proceedings of the 14th International Conference on Mining Software Repositories*. IEEE Press, 2017, pp. 463–466.

[209] D. Spencer, *Card sorting: Designing usable categories*. Rosenfeld Media, 2009.

[210] G. E. Box, G. M. Jenkins, G. C. Reinsel, and G. M. Ljung, *Time series analysis: forecasting and control*. John Wiley & Sons, 2015.

[211] W. Lidwell, K. Holden, and J. Butler, *Universal Principles of Design, Revised and Updated: 125 Ways to Enhance Usability, Influence Perception, Increase Appeal, Make Better Design Decisions, and Teach through Design*, 2nd ed. Rockport Publishers, January 2010.

[212] L. Pascarella, F.-X. Geiger, F. Palomba, D. D. Nucci, I. Malavolta, and A. Bacchelli, "Self-Reported Activities of Android Developers," in *5th IEEE/ACM International Conference on Mobile Software Engineering and Systems*. New York, NY: ACM, May 2018, p. to appear. [Online]. Available: http://www.ivanomalavolta.com/files/papers/mobilesoft_2018_self.pdf

[213] C. J. Kapser and M. W. Godfrey, ""cloning considered harmful" considered harmful: patterns of cloning in software," *Empirical Software Engineering*, vol. 13, no. 6, p. 645, 2008.

[214] H. Bagheri, J. Garcia, A. Sadeghi, S. Malek, and N. Medvidovic, "Software architectural principles in contemporary mobile software: from conception to pratice," *The Journal of Systems and Software*, vol. 119, no. 1, pp. 31–44, 2016. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2016.05.039

[215] B. J. Luijten, "The influence of software maintainability on issue handling," Master's thesis, Faculty EEMCS, Delft University of Technology, 2009.

[216] D. Athanasiou, A. Nugroho, J. Visser, and A. Zaidman, "Test code quality and its relation to issue handling performance," *IEEE Transactions on Software Engineering*, vol. 40, no. 11, pp. 1100–1125, 2014. [Online]. Available: http://ieeexplore.ieee.org/document/6862882/

[217] T. L. Alves, "Determination of number of clusters in K-means clustering and application in colour segmentation," in *Proceedings of Simulation and EGSE facilities for Space Programmes (SESP2010)*, 2010.

[218] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. D. Penta, A. D. Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad," in *ICSE '15 Proceedings of the 37th International Conference on Software Engineering*, I. P. Piscataway, Ed. IEEE, 2015, pp. 403–414.

[219] S. Koch, "Software evolution in open source projects - a large-scale investigation," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 19, no. 6, pp. 361–382, 2007. [Online]. Available: http://dx.doi.org/10.1002/smr.348

[220] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "Api change and fault proneness: a threat to the success of android apps," in *Proceedings of the 2013 9th joint meeting on foundations of software engineering*. ACM, 2013, pp. 477–487.

[221] H. Bagheri, J. Garcia, A. Sadeghi, S. Malek, and N. Medvidovic, "Software architectural principles in contemporary mobile software: from conception to practice," *Journal of Systems and Software*, vol. 119, pp. 31–44, 2016.

[222] "Global mobile OS market share in sales to end users from 1st quarter 2009 to 1st quarter 2018," 2018. [Online]. Available: https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/

[223] "Global developer population and demographic study 2017," 2017. [Online]. Available: https://evansdata.com/press/viewRelease.php?pressID=244

[224] "Number of available applications in the Google Play Store from December 2009 to June 2018," 2018. [Online]. Available: https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/

[225] ISO/IEC, "ISO/IEC 25010 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models," Tech. Rep., 2010.

[226] V. Garousi, M. Felderer, and M. V. Mäntylä, "The need for multivocal literature reviews in software engineering: complementing systematic literature reviews with grey literature," in *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*. ACM, 2016, p. 26.

[227] D. Budgen and P. Brereton, "Performing systematic literature reviews in software engineering," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 1051–1052.

[228] V. Garousi, M. Felderer, and M. V. Mäntylä, "Guidelines for including grey literature and conducting multivocal literature reviews in software engineering," *Information and Software Technology*, 2018.

[229] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in *Proceedings of the 18th international conference on evaluation and assessment in software engineering*. ACM, 2014, p. 38.

[230] B. Kitchenham, R. Pretorius, D. Budgen, O. P. Brereton, M. Turner, M. Niazi, and S. Linkman, "Systematic literature reviews in software engineering–a tertiary study," *Information and software technology*, vol. 52, no. 8, pp. 792–805, 2010.

[231] R. C. Martin, *Clean architecture: a craftsman's guide to software structure and design*. Prentice Hall Press, 2017.

[232] J. Gilbert and C. Stoll, "Architecting iOS Apps with VIPER." [Online]. Available: https://www.objc.io/issues/13-architecture/viper/

[233] E. Wolff, *Microservices: flexible software architecture*. Addison-Wesley Professional, 2016.

[234] A. I. Wasserman, "Software engineering issues for mobile application development," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*. ACM, 2010, pp. 397–400.

[235] E. Campos, U. Kulesza, R. Coelho, R. Bonifácio, and L. Mariano, "Unveiling the Architecture and Design of Android Applications," in *Proceedings of the 17th International Conference on Enterprise Information Systems-Volume 2*, 2015, pp. 201–211.

[236] K. Sokolova, M. Lemercier, L. Garcia, and L. C. Saint Luc, "Towards High Quality Mobile Applications: Android Passive MVC Architecture," *International Journal On Advances in Software*, vol. 7, no. 2, pp. 123–138, 2014.

[237] T. Dürschmid, M. Trapp, and J. Döllner, "Towards architectural styles for Android app software product lines," in *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*. IEEE Press, 2017, pp. 58–62.

[238] M. E. Joorabchi, A. Mesbah, and P. Kruchten, "Real challenges in mobile app development," in *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*. IEEE, 2013, pp. 15–24.

[239] V. Garousi and B. Küçük, "Smells in software test code: A survey of knowledge in industry and academia," *Journal of Systems and Software*, vol. 138, pp. 52–81, 2018.

[240] N. Tripathi, P. Seppänen, G. Boominathan, M. Oivo, and K. Liukkunen, "Insights into startup ecosystems through exploration of multi-vocal literature," *Information and Software Technology*, vol. 105, pp. 56–77, 2019.

[241] J. Soldani, D. A. Tamburri, and W.-J. Van Den Heuvel, "The pains and gains of microservices: A systematic grey literature review," *Journal of Systems and Software*, vol. 146, pp. 215–232, 2018.

[242] S. Maro, J.-P. Steghöfer, and M. Staron, "Software traceability in the automotive domain: Challenges and solutions," *Journal of Systems and Software*, vol. 141, pp. 85–110, 2018.

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