Building and evaluating a theory of architectural technical debt in software-intensive systems☆

Roberto Verdecchia [a], Philippe Kruchten [b], Patricia Lago [a,c], Ivano Malavolta [a,*]

[a] *Vrije Universiteit Amsterdam, The Netherlands*
[b] *University of British Columbia, Vancouver, Canada*
[c] *Chalmers University of Technology, Gothenburg, Sweden*

ARTICLE INFO

ABSTRACT

Architectural technical debt in software-intensive systems is a metaphor used to describe the "big" design decisions (*e.g.,* choices regarding structure, frameworks, technologies, languages, etc.) that, while being suitable or even optimal when made, significantly hinder progress in the future. While other types of debt, such as code-level technical debt, can be readily detected by static analyzers, and often be refactored with minimal or only incremental efforts, architectural debt is hard to be identified, of wide-ranging remediation cost, daunting, and often avoided.

In this study, we aim at developing a better understanding of how software development organizations conceptualize architectural debt, and how they deal with it. In order to do so, in this investigation we apply a mixed empirical method, constituted by a grounded theory study followed by focus groups. With the grounded theory method we construct a theory on architectural technical debt by eliciting qualitative data from software architects and senior technical staff from a wide range of heterogeneous software development organizations. We applied the focus group method to evaluate the emerging theory and refine it according to the new data collected.

The result of the study, *i.e.,* a theory emerging from the gathered data, constitutes an encompassing conceptual model of architectural technical debt, identifying and relating concepts such as its symptoms, causes, consequences, management strategies, and communication problems. From the conducted focus groups, we assessed that the theory adheres to the four evaluation criteria of classic grounded theory, *i.e.,* the theory *fits* its underlying data, is able to *work*, has *relevance*, and is *modifiable* as new data appears.

By grounding the findings in empirical evidence, the theory provides researchers and practitioners with novel knowledge on the crucial factors of architectural technical debt experienced in industrial contexts.

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 (Cunningham, 1992), but it only got some real attention from researchers in the last 10 years (Brown et al., 2010). 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" (Avgeriou et al., 2016).

Technical debt can take many different forms in software development, and can be found in many different places (Kruchten et al., 2012). 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 (Verdecchia et al., 2020a) we reported a preliminary version of our theory for ATD. In this paper we extended our previous work in multiple ways:

- we expanded the theory by including a total of 9 categories and 63 concepts (2 and 24 additional ones with respect to Verdecchia et al., 2020a), 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 3 reports the results of our investigation, with each of the Sections 3.1-3.10 dedicated to the description of a specific category of our theory. Related work and theory evaluation results are reported in Section 4 and Section 5, respectively. Threats to validity to our study are reported in Section 6. Section 7 concludes the paper.

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 (A) - (H) of Fig. 1). Afterwards, once the theory on ATD was established, we applied focus groups in order to evaluate and refine our theory (see step (I) of Fig. 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 2.1 summarizes the grounded theory method, Section 2.2 documents the grounded theory design and execution, including the details about data collection and data analysis, and Section 2.3 details the focus group method adopted to evaluate and complement the emerging theory.

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 and Stern (2001), 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 Fig. 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:

(A) **Theoretical Sampling.** New data is collected iteratively by purposely identifying current gaps and/or unsaturated categories of the theory. Theoretical sampling guides the selection of new data sources (*e.g.,* participants), and the data to be collected (*e.g.,* by generating iteratively interview questions).

(B) **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.
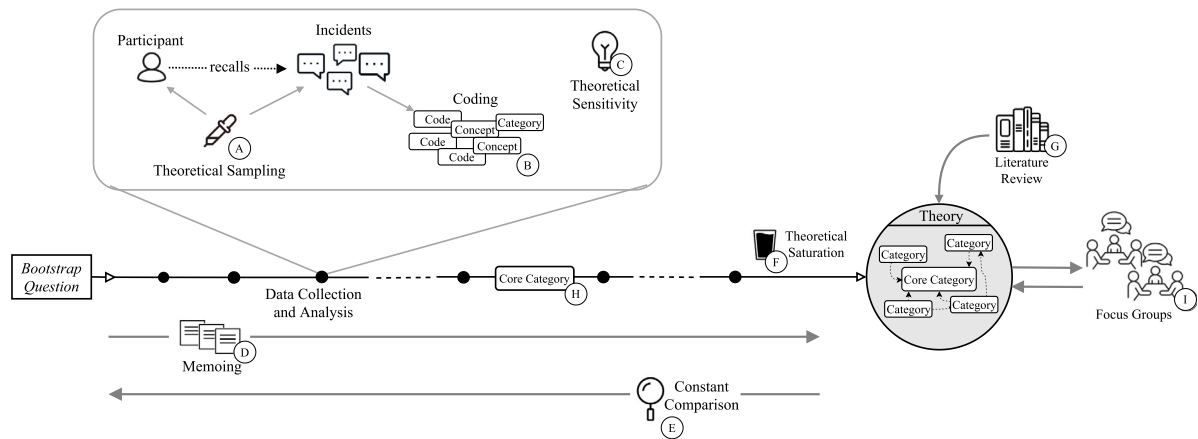
Fig. 1. Overview of the grounded theory research method ((Ⓐ) - (Ⓗ)), evaluated via focus groups ((Ⓘ)).

Ⓒ **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 Kenny and Fourie, 2015).

2.2. Grounded theory design and execution

For this study, we adopted the classic "Glaserian" method (Glaser and Strauss, 1967), 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 (2005), as explained in Section 2.2.2. In divergence to the other GT stances, during the analysis process of the method described by Glaser (2005), a "core category" is established. The core category captures the most variation in the data (Glaser, 1978b) while addressing the main concern of the study participants (see item Ⓗ of Fig. 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 first author 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" (Glaser, 1978a). As prescribed by Glaser (Glaser, 1992), 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. (2016), 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.

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 fitted participants for data collection, given their hands-on experience on a vast range of ongoing (and concluded) long-lived software projects. Table 1 presents an overview of the participant demographics. Interviews lasted approximately 1 h 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" (Adolph et al., 2011) of GT by starting our first interview with the bootstrapping question described in Section 2.1. Then, the other interview questions emerged iteratively by following theoretical sampling, in order

Table 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.

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" (Rubin and Rubin, 2011), *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 (1994), 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 (Morse, 1994).

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 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 (Oliver et al., 2005).[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 (Glaser, 1978a). Fig. 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.

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 (Glaser and Strauss, 1967; Glaser, 1978a). Specifically, we examined the whole body of text transcripts, subdivided them into separate incidents
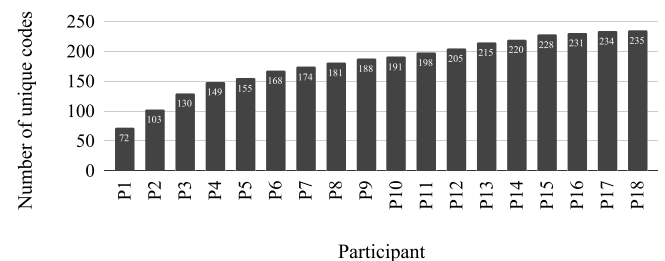

Fig. 2. Cumulative unique number of codes per participant, showing theoretical saturation.

(Glaser and Strauss, 1967), 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, 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 Fig. 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.,* Adolph et al. (2011) and Hoda and Noble (2017). 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 Fig. 3.

During the entirety of the coding procedures, we made use of *memoing* (Glaser and Strauss, 1967). We created textual memos to elaborate concepts (i) related to single incidents (*e.g., "This*

---

[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 lengthy transcripts, which would have impacted negatively the subsequent data analysis.
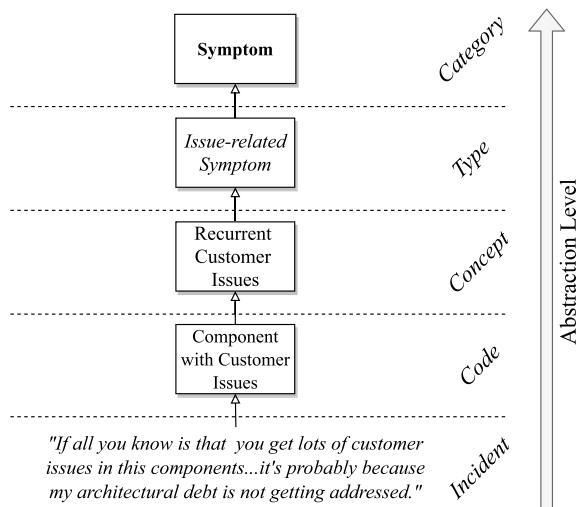
Fig. 3. Example of Abstraction Levels of the ATD Theory.

Fig. 4. Example of memo used to gain a summary overview of the codes emerging from a single interview.

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"*). 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 Fig. 4.

As described in Section 2.2.1, we analyzed 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 2.2.1).

It should be noted that numerous concepts included in our theory possess a multifaceted 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 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 Engward (2013).

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 analyzed and reviewed the obtained results through several iterations.

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 (Kontio et al., 2008) (item ① in Fig. 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 evaluation of the theory was conducted by following the criteria characteristic of the Glaserian GT method (Glaser, 1978a), 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.

As shown in Fig. 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 Fig. 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 (Fig. 5, Step 3a), followed by a discussion among the participants about the topic presented (Fig. 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
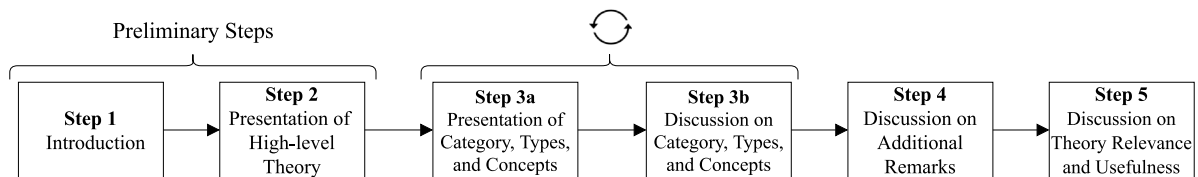
Preliminary Steps

| Step 1 Introduction | Step 2 Presentation of High-level Theory | Step 3a Presentation of Category, Types, and Concepts | Step 3b Discussion on Category, Types, and Concepts | Step 4 Discussion on Additional Remarks | Step 5 Discussion on Theory Relevance and Usefulness |

Fig. 5. Main steps of the focus group sessions.

Table 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.

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 (Fig. 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 (Fig. 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 (Lomborg and Kirkevold, 2003). 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 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 (Bryman, 2001). Focus group sessions lasted approximately 1.5 h, 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 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 (⚎).

3. A theory of architectural technical debt

Fig. 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" (ISO/IEC/IEEE, 2011). 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 16 162 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" (Avgeriou et al., 2016).

In addition to reporting the categories of our 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 (Glaser, 1978a; Strauss and Corbin, 1998).

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 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 the other ones reported in Fig. 6.

At the root of each ATD item lies one or more ***cause***. Each cause can *generate* one or more items (see Section 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]
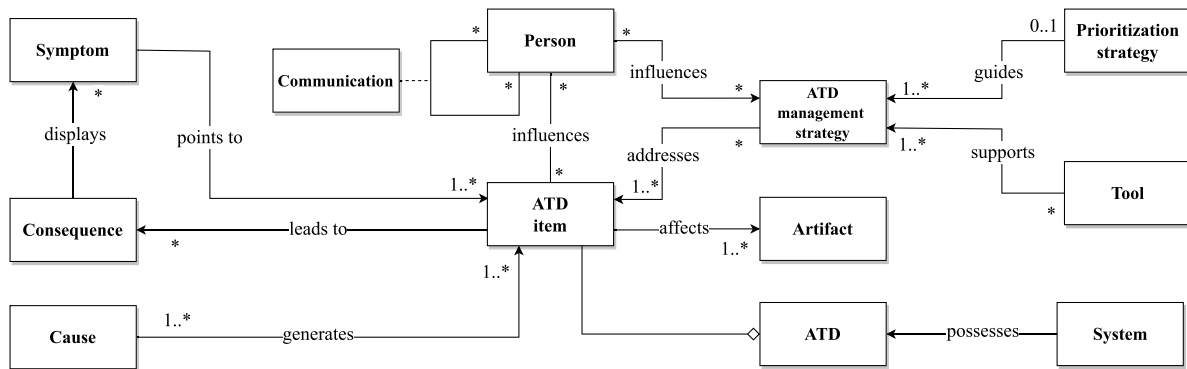
Fig. 6. Core categories of the ATD theory and their relations (Verdecchia et al., 2020a).

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 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 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 (Kruchten, 2008) (see Section 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 dependency 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 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 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 3.1–3.10, 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*).

---

2 https://clang.llvm.org/extra/clang-tidy.
3 https://www.sonarqube.org.
4 https://www.ndepend.com/.

3.1. ATD items

An overview of the ATD items residing in software-intensive systems which emerged in our theory is depicted in Fig. 7. The relation between elements of Fig. 7 has to be interpreted as a "is a type of" relation (same applies for Figs. 8–11, and Fig. 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, become of architectural relevance. The reminder of this section is dedicated to the description of each concept belonging to the ATD Item category.
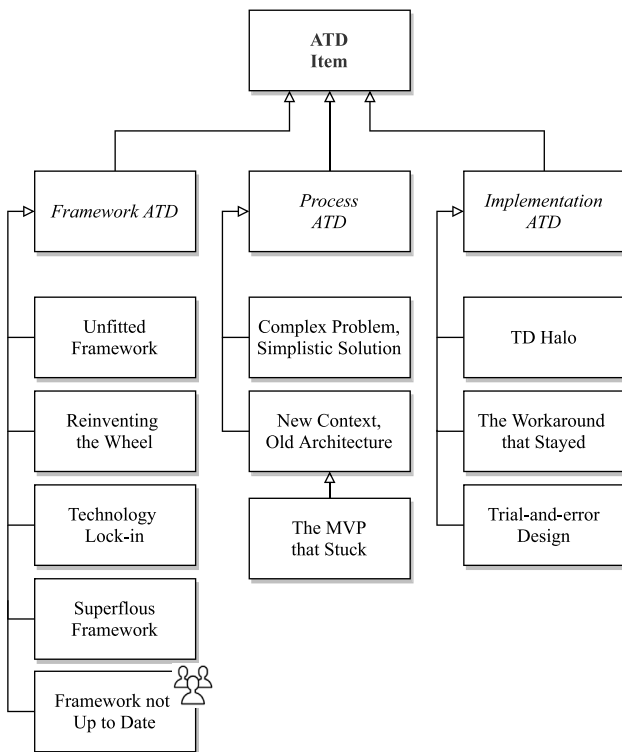

Fig. 7. Overview of emerging ATD items.

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, 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 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 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:

---

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

"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 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" (Zerouali et al., 2018), 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.

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 (Almonaies et al., 2010), may need to evolve at a much lower pace than mobile apps, which are generally characterized by a rapidly changing ecosystem (Verdecchia et al., 2019).

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.

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.

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 Fig. 8.
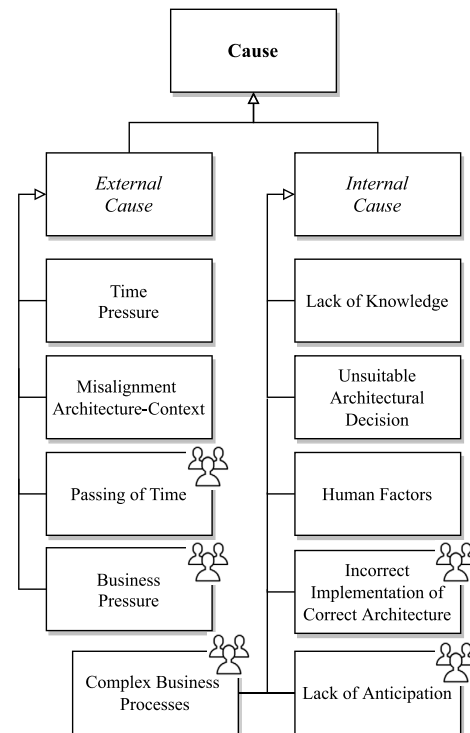

Fig. 8. Overview of ATD causes.

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 3.8 and 3.9.

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]

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 Kruger and Dunning, 1999). Due to the importance of this topic in our theory, we further discuss findings related to human factors in Section 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 (Conway, 1968) 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.

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 Fig. 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.
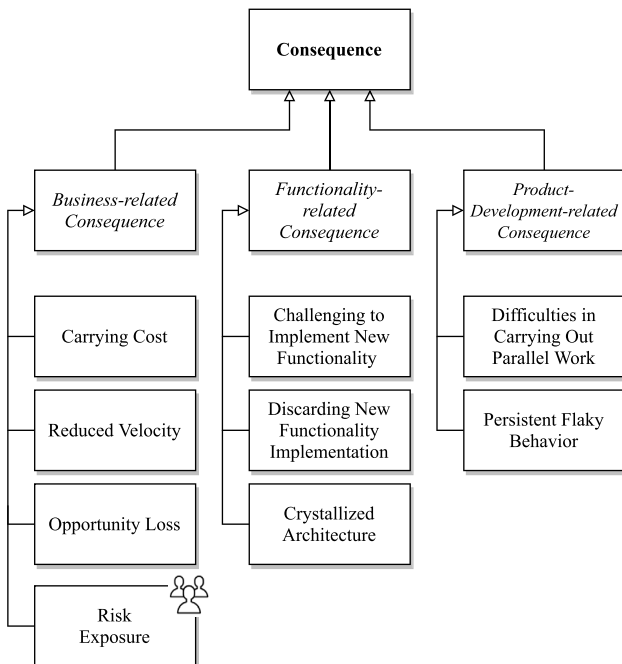

Fig. 9. Overview of ATD consequences.

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 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 accommodate 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 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 lose 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.

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 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]

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 (Wert et al., 2014). 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 crosscutting 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 Behavior. Software-intensive systems afflicted by a severe amount of ATD can become unpredictable in terms of expected behavior. 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 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]

3.4. Symptoms

An overview of the ATD symptoms is presented in Fig. 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.
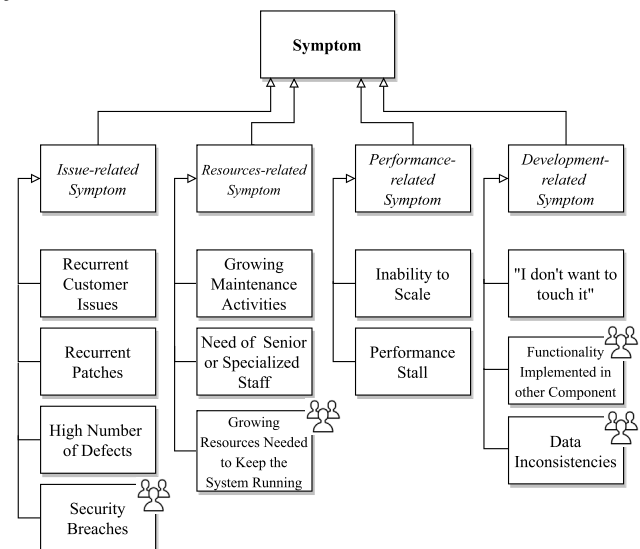
Fig. 10. Overview of ATD symptoms.

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 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.

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 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 Sections 3.9 and 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 (Mateos et al., 2019), 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]

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]

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.

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 Fig. 11, and further described in the reminder of this section.
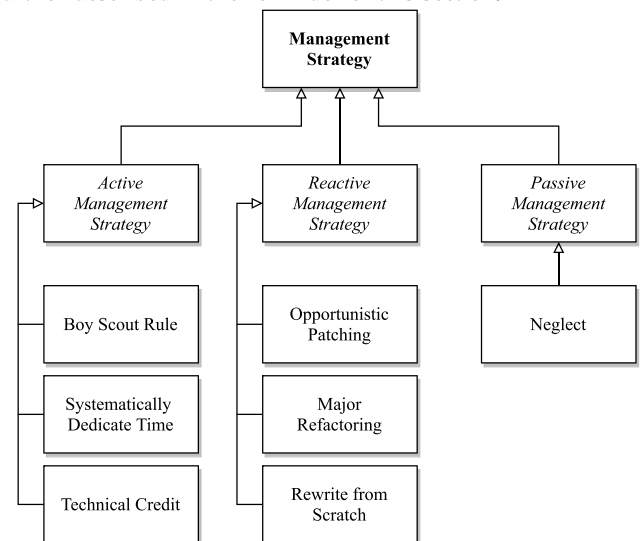

Fig. 11. Overview of ATD management strategies.

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 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]

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 (Chiang and Bayrak, 2006), *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]

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.

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.

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 Fig. 12 and described below.
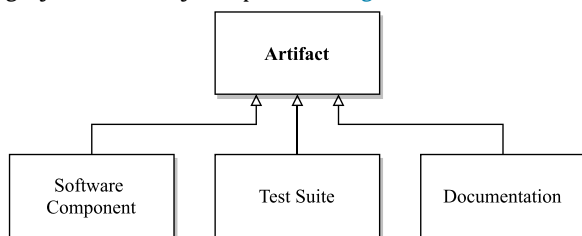

Fig. 12. Overview of ATD concepts related to *artifact*.

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:

---

[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/.

"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 artifact 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]

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 (2008), 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 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]

3.9. Person

This category deals with concepts related to the human nature of software professionals. As can be deduced from Sections 3.4 and 3.8 , people can support the discovery and prioritization of
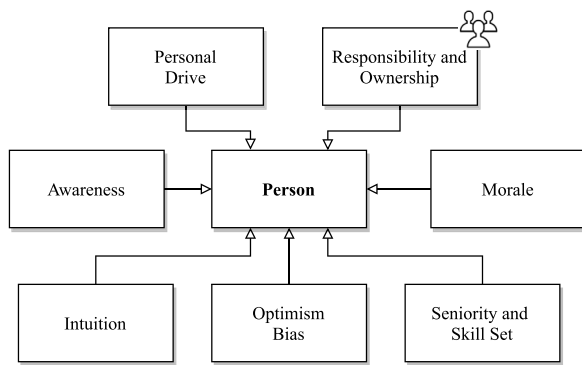
Fig. 13. Overview of ATD concepts related to *person*.

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 Fig. 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 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 optimism one, such as wishful thinking, self-serving bias (Myers and Smith, 2015), and the Dunning–Kruger effect (Kruger and Dunning, 1999). Such biases notably lead to the emergence of the planning fallacy phenomenon (Kahneman and Tversky, 1977), 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]

3.10. Communication

We identified 4 main concepts related to the *communication* category, namely *exposition*, *impediments*, *blame*, and *communication with stakeholders*, as depicted in Fig. 14 and described in the following.
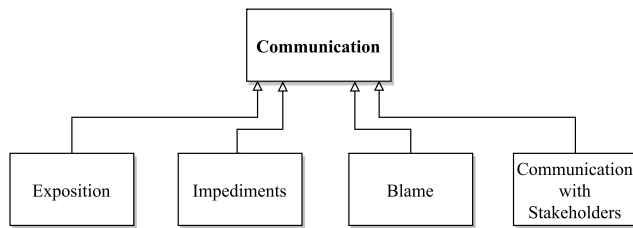
Fig. 14. Overview of ATD concepts related to *communication*.

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]

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. Related work

As recommended by Glaserian GT principles (Glaser and Strauss, 1967), 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 (Martini and Bosch, 2017; Martini et al., 2015; Besker et al., 2018; Li et al., 2016). 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 3.

Note that concepts identified in the literature which emerge in our theory under a different category (*e.g.*, in Martini et al., 2015 "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 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.

Martini and Bosch (2017) 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 2 of this study and Section 2 of Martini and Bosch, 2017). Regarding the results, Martini and Bosch (2017) presents a taxonomy of ATD items and a model of their effects: the specific *ATD items* reported in Table 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 (Martini et al., 2015) 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 Martini and Bosch (2017) and Martini et al. (2015) 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 (Urquhart et al., 2010). 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. (2018) 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 Besker et al. (2018), such as *ATD detection*, *ATD identification*, *ATD measurement*, *ATD monitoring* and related concepts, did not emerge in our theory. Rather than

Table 3
Complementary concepts to our theory identified in the literature.

Source	Concept	Type	Category	Definition
Martini and Bosch (2017)	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 and Bosch (2017)	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 and Bosch (2017)	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 and Bosch (2017)	Superfluous testing	Product development	Consequence	Unnecessary tests performed due to ATD.
Martini et al. (2015)	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. (2015)	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. (2015)	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. (2015)	Non-completed refactoring	Internal	Cause	Non-completed refactoring activity, introducing new ATD.
Besker et al. (2018)	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. (2018)	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.

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* Verdecchia et al., 2018), 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* (Verdecchia et al., 2018; Li et al., 2015), such as ATD related to component dependency (Roveda et al., 2018) or modularity (Li et al., 2014). 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' popular book "The Mythical Man Month" (Brooks, 1995), 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" (Fowler, 2018), 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 (Szyperski et al., 2002), 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 (Verdecchia et al., 2018; Li et al., 2015; Martini and Bosch, 2015a), and are mapped to our category *new context, old architecture*. In Martini and Bosch (2015b), 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 3.8). The literature further investigates other emerging categories, such as TD management strategies (Alves et al., 2016), and the impact of TD on morale (Ghanbari et al., 2017), but does not systematically focus on the architectural level as we do.

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 2.3. Specifically, we base the evaluation of our theory on the four criteria presented by Glaser (1978a), as we followed such GT stance to construct our theory. In the following, the assessment results of each evaluation criterion is discussed separately.

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

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.

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.

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 Figs. 7–13), 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.

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.,* Hoda and Noble, 2017), the verifiability of our results should derive from the soundness of the research method followed. We therefore provide in Section 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 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 (Glaser and Strauss, 1967). 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.

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 Verdecchia et al. (2020b). 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, 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.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

References

Adolph, S., Hall, W., Kruchten, P., 2011. Using grounded theory to study the experience of software development. Empir. Softw. Eng. 16 (4), 487–513. http://dx.doi.org/10.1007/s10664-010-9152-6.

Almonaies, A.A., Cordy, J.R., Dean, T.R., 2010. Legacy system evolution towards service-oriented architecture. In: International Workshop on SOA Migration and Evolution. pp. 53–62. http://dx.doi.org/10.4018/978-1-4666-2488-7.ch003.

Alves, N., Mendes, T.S., de Mendonça, M.G., Spínola, R.O., Shull, F., Seaman, C., 2016. Identification and management of technical debt: A systematic mapping study. Inf. Softw. Technol. 70, 100–121. http://dx.doi.org/10.1016/j.infsof.2015.10.008.

Avgeriou, P., Kruchten, P., Ozkaya, I., Seaman, C., 2016. Managing Technical Debt in Software Engineering. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, http://dx.doi.org/10.4230/DagRep.6.4.110,

Besker, T., Martini, A., Bosch, J., 2018. Managing architectural technical debt: A unified model and systematic literature review. J. Syst. Softw. 135, http://dx.doi.org/10.1016/j.jss.2017.09.025.

Brooks, Jr., F.P., 1995. The Mythical Man-Month (Anniversary Edition). Addison Wesley, Boston.

Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., Lim, E., MacCormack, A., Nord, R., Ozkaya, I., Sangwan, R., Seaman, C.B., Sullivan, K., Zazworka, N., 2010. Managing technical debt in software-reliant systems. In: Future of Software Engineering Research. pp. 47–52. http://dx.doi.org/10.1145/1882362.1882373.

Bryman, A., 2001. Social Research Methods. Oxford University Press, ISBN: 9780199689453, pp. 365–399.

Chiang, C.-C., Bayrak, C., 2006. Legacy software modernization. In: 2006 IEEE International Conference on Systems, Man and Cybernetics. 2, IEEE, pp. 1304–1309. http://dx.doi.org/10.1109/ICSMC.2006.384895.

Conway, M.E., 1968. How do committees invent. Datamation 14 (4), 28–31.

Cunningham, W., 1992. The wycash portfolio management system. In: OOPSLA Proceedings. http://dx.doi.org/10.1145/157710.157715.

Engward, H., 2013. Understanding grounded theory. Nurs. Stand. 28 (7), http://dx.doi.org/10.7748/ns2013.10.28.7.37.e7806.

Fowler, M., 2018. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional.

Ghanbari, H., Besker, T., Martini, A., Bosch, J., 2017. Looking for peace of mind?: manage your (technical) debt: an exploratory field study. In: ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. http://dx.doi.org/10.1109/ESEM.2017.53.

Glaser, B., 1978a. Theoretical Sensitivity. Sociology Press.

Glaser, B., 1978b. Theoretical sensitivity. In: Advances in the Methodology of Grounded Theory.

Glaser, B., 1992. Basics of Grounded Theory Analysis: Emergence Vs Forcing. Sociology press.

Glaser, B., 2005. The Grounded Theory Perspective III: Theoretical Coding. Sociology Press.

Glaser, B., Strauss, A., 1967. Discovery of Grounded Theory: Strategies for Qualitative Research. Aldine.

Hoda, R., Noble, J., 2017. Becoming agile: a grounded theory of agile transitions in practice. In: International Conference on Software Engineering. IEEE Press, http://dx.doi.org/10.1109/ICSE.2017.21.

ISO/IEC/IEEE, 2011. Systems and software engineering – architecture description. In: ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000). pp. 1–46. http://dx.doi.org/10.1109/IEEESTD.2011.6129467.

Kahneman, D., Tversky, A., 1977. Intuitive Prediction: Biases and Corrective Procedures. Technical Report, Cambridge University Press, http://dx.doi.org/10.1017/CBO9780511809477.031.

Kenny, M., Fourie, R., 2015. Contrasting classic, straussian, and constructivist grounded theory: methodological and philosophical conflicts. Qual. Rep. 20 (8), 1270–1289.

Kontio, J., Bragge, J., Lehtola, L., 2008. The focus group method as an empirical tool in software engineering. In: Guide to Advanced Empirical Software Engineering. Springer, pp. 93–116. http://dx.doi.org/10.1007/978-1-84800-044-5.

Kruchten, P., 2008. What colour is your backlog? Available Online: https://tinyurl.com/y6f7vhpx (Accessed 10th May 2020).

Kruchten, P., Nord, R., Ozkaya, I., 2012. Technical debt: from metaphor to theory and practice. IEEE Softw. 29 (6), 18–21. http://dx.doi.org/10.1109/MS.2012.167.

Kruger, J., Dunning, D., 1999. Unskilled and unaware of it: how difficulties in recognizing one's own incompetence lead to inflated self-assessments.. J. Pers. Soc. Psychol. 77 (6), 1121. http://dx.doi.org/10.1037//0022-3514.77.6.1121.

Li, Z., Avgeriou, P., Liang, P., 2015. A systematic mapping study on technical debt and its management. J. Syst. Softw. 101, 193–220. http://dx.doi.org/10.1016/j.jss.2014.12.027.

Li, Z., Liang, P., Avgeriou, P., 2016. Architecture viewpoints for documenting architectural technical debt. In: Software Quality Assurance. Elsevier, pp. 85–132. http://dx.doi.org/10.1016/B978-0-12-802301-3.00005-3.

Li, Z., Liang, P., Avgeriou, P., Guelfi, N., Ampatzoglou, A., 2014. An empirical investigation of modularity metrics for indicating architectural technical debt. In: International ACM Conference on Quality of Software Architectures. http://dx.doi.org/10.1145/2602576.2602581.

Lomborg, K., Kirkevold, M., 2003. Truth and validity in grounded theory–a reconsidered realist interpretation of the criteria: fit, work, relevance and modifiability. Nurs. Phil. 4 (3), 189–200. http://dx.doi.org/10.1046/j.1466-769x.2003.00139.x.

Martini, A., Bosch, J., 2015a. The danger of architectural technical debt: Contagious debt and vicious circles. In: Working IEEE/IFIP Conference on Software Architecture. IEEE, pp. 1–10. http://dx.doi.org/10.1109/WICSA.2015.31.

Martini, A., Bosch, J., 2015b. Towards prioritizing architecture technical debt: information needs of architects and product owners. In: Euromicro Conference on Software Engineering and Advanced Applications. IEEE, pp. 422–429. http://dx.doi.org/10.1109/SEAA.2015.78.

Martini, A., Bosch, J., 2017. On the interest of architectural technical debt: uncovering the contagious debt phenomenon. J. Softw. Evol. Process http://dx.doi.org/10.1002/smr.1877.

Martini, A., Bosch, J., Chaudron, M., 2015. Investigating architectural technical debt accumulation and refactoring over time: A multiple-case study. Inf. Softw. Technol. 67, 237–253. http://dx.doi.org/10.1016/j.infsof.2015.07.005.

Mateos, C., Zunino, A., Flores, A., Misra, S., 2019. COBOL systems migration to SOA: assessing antipatterns and complexity. Inform. Technol. Control 48 (1), 71–89. http://dx.doi.org/10.5755/j01.itc.48.1.21566.

Morse, J.M., 1994. "Emerging from the data": The cognitive processes of analysis in qualitative inquiry. Crit. Issues Qqual. Res. Methods 23–46.

Myers, D., Smith, S., 2015. Exploring Social Psychology. McGraw-Hill.

Oliver, D., Serovich, J., Mason, T., 2005. Constraints and opportunities with interview transcription: Towards reflection in qualitative research. Soc. Forces 1273. http://dx.doi.org/10.1353/sof.2006.0023.

Rose, K., 1994. Unstructured and semi-structured interviewing. Nurse Res. 1 (3), 23–32. http://dx.doi.org/10.7748/nr.1.3.23.s4.

Roveda, R., Fontana, F.A., Pigazzini, I., Zanoni, M., 2018. Towards an architectural debt index. In: Euromicro Conference on Software Engineering and Advanced Applications. IEEE, http://dx.doi.org/10.1109/SEAA.2018.00073.

Rubin, H.J., Rubin, I.S., 2011. Qualitative Interviewing: The Art of Hearing Data. Sage Publications, pp. 58–59.

Schreiber, R.S., Stern, P.N., 2001. Using Grounded Theory in Nursing. Springer Publishing Company.

Stol, K.-J., Ralph, P., Fitzgerald, B., 2016. Grounded theory in software engineering research: a critical review and guidelines. In: IEEE/ACM International Conference on Software Engineering. http://dx.doi.org/10.1145/2884781.2884833.

Strauss, A., Corbin, J., 1998. Basics of Qualitative Research Techniques. Sage publications Thousand Oaks.

Szyperski, C., Gruntz, D., Murer, S., 2002. Component Software: Beyond Object-Oriented Programming. Pearson Education.

Urquhart, C., Lehmann, H., Myers, M.D., 2010. Putting the 'theory' back into grounded theory: guidelines for grounded theory studies in information systems. Inform. Syst. J. 20 (4), 357–381. http://dx.doi.org/10.1111/j.1365-2575.2009.00328.x.

Verdecchia, R., Kruchten, P., Lago, P., 2020a. Architectural technical debt: A grounded theory. In: European Conference on Software Architecture. Springer, pp. 202–219. http://dx.doi.org/10.1007/978-3-030-58923-3_14.

Verdecchia, R., Lago, P., Malavolta, I., Ozkaya, I., 2020b. ATDx: Building an architectural technical debt index.. In: International Conference on Evaluation of Novel Approaches to Software Engineering. pp. 531–539. http://dx.doi.org/10.5220/0009577805310539.

Verdecchia, R., Malavolta, I., Lago, P., 2018. Architectural technical debt identification: The research landscape. In: IEEE/ACM International Conference on Technical Debt. IEEE, pp. 11–20. http://dx.doi.org/10.1145/3194164.3194176.

Verdecchia, R., Malavolta, I., Lago, P., 2019. Guidelines for architecting android apps: A mixed-method empirical study. In: 2019 IEEE International Conference on Software Architecture (ICSA). IEEE, pp. 141–150. http://dx.doi.org/10.1109/ICSA.2019.00023.

Wert, A., Oehler, M., Heger, C., Farahbod, R., 2014. Automatic detection of performance anti-patterns in inter-component communications. In: International ACM Conference on Quality of Software Architectures. http://dx.doi.org/10.1145/2602576.2602579.

Zerouali, A., Constantinou, E., Mens, T., Robles, G., González-Barahona, J., 2018. An empirical analysis of technical lag in npm package dependencies. In: International Conference on Software Reuse. Springer, pp. 95–110. http://dx.doi.org/10.1007/978-3-319-90421-4_6.