Architectural Technical Debt: A Grounded Theory

Roberto Verdecchia[1], Philippe Kruchten[2], and Patricia Lago[1]

[1] Vrije Universiteit Amsterdam, The Netherlands, {r.verdecchia|p.lago}@vu.nl
[2] University of British Columbia, Vancouver, Canada, pbk@ece.ubc.ca

Abstract. Architectural technical debt in a software-intensive system is driven by design decisions about its structure, frameworks, technologies, languages, etc. Unlike code-level technical debt, which can be readily detected by static analysers, and can often be refactored with minimal efforts, architectural debt is hard to detect, and its remediation is wide-ranging, daunting, and often avoided. The objective of this study is to develop a better understanding of how software development organisations conceptualize their architectural debt, and how they deal with it, if at all. We used a grounded theory method, eliciting qualitative data from software architects and senior technical staff from a wide range of software development organizations. The result of the study, i.e., the theory emerging from the collected data, constitutes an encompassing conceptual theory of architectural debt, identifying and relating concepts such as symptoms, causes, consequences, and management strategies. By grounding the findings in empirical data, the theory provides researchers and practitioners with evidence of which crucial factors of architectural technical debt are experienced in industrial contexts.

Keywords: Software Architecture · Technical Debt · Grounded Theory.

1 Introduction

Quoting Avgeriou et al. [3], "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".

Technical Debt (TD) can take many different forms in software development, and can be found in many different places [16]. While much of the literature and tooling available today address code-level TD, our focus is on Architectural Technical Debt (ATD). This is the technical debt incurred at the architectural level of software design, i.e., in the decisions related to structure (layering, decomposition in subsystems, interfaces), technologies (frameworks, packages, libraries, deployment approach), or even languages, development process, and platform. As software systems grow in size and their lifespan extends, many of these original design choices become constraints, limiting future evolution or even preventing it. To evolve the system, developers find workarounds, introducing 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.

To characterize ATD, find attributes of ATD, and develop an interpretation of ATD based on empirical evidence, we used a grounded theory approach [12] with experienced industry practitioners as subjects. The result of our study is an ATD "theory", providing empirical evidence of how software development practitioners conceptualize ATD and its management. Some of our theory results can also be applied to other forms of technical debt, such as code-level TD.

2 Research Method

For our study, we adopted the classic "Glaserian" Grounded Theory (GT) method [12], and we stayed with it throughout the whole study, from data collection, to data analysis and synthesis, with the exception of our adoption of a different "coding family" w.r.t. the ones suggested by Glaser [11], as explained in Section 2.2. This GT approach has given us a fresh and independent viewpoint on ATD, by letting concepts emerge from the personal experience of our participants, rather than the preconceived views of the researchers. In line with GT principles, we delayed the review of the literature until after our theory emerged, in order to avoid the influence of existing concepts on our theory [10]. Specifically, the first author was not too immersed in the TD world prior to this study, and refrained from conducting an extensive literature review on ATD before analyzing the data, minimizing possible confirmation biases, and improving his "theoretical sensitivity" [9]. In fact, as stated by Glaser et al., prior knowledge "violates the basic premise of GT - that the theory emerges from the data, not from extant theory" [13]. We also followed the recommendations of Stol et al. [27], on the application of GT to software engineering topics, and avoided the typical pitfalls they have identified. The investigation, including data collection, data analysis, and reporting, lasted approximately 6 months.

2.1 Data Collection

To collect data, we conducted semi-structured interviews with industrial practitioners. Initial participants were recruited by convenience and then subsequent ones driven by theoretical sampling [10], that is, tactically picking new subjects that would allow to confirm or disconfirm the findings so far, or to explore new areas. Specifically, the initial participants were contacted within our personal network. Subsequent participants were selected by following theoretical sampling, in order to fill the gaps identified in our emerging theory, and/or to explore unsaturated concepts [10]. Specifically, we identified via theoretical sampling [12] 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. We interviewed 18 experienced software practitioners, with a mean industrial experience of 17.5 years, from 14 distinct companies in different industrial domains. Table 1 presents an overview of the participant demographics. Interviews lasted approximately 1 hour and were conducted face-to-face at the practitioners' workplaces or, when not possible, via video-calls.

As the emerging theory should guide the sampling process, we solved the "bootstrap problem" [1] of GT by starting our first interview with the question: *"Which architectural design decision do you regret the most today?"*. Subsequently,

Table 1. Participant Demographics

ID	Role	Ex	Domain	OS	CC
P1	Senior Vice-President of SE	21	Banking	S	72
P2	Software Staff Engineer	17	Telecom	M	103
P3	Senior Director of SE	20	Enterprise Software	XL	130
P4	Chief Technology Officer	14	Financial Services	M	149
P5	Senior Software Engineer	22	Health	L	155
P6	Senior Software Engineer	8	Software Tooling	M	168
P7	Senior Software Engineer	18	Software Tooling	M	174
P8	Senior Software Engineer	23	Software Tooling	M	181
P9	Vice-President of Product	15	Data Analysis	M	188
P10	Senior Software Engineer	12	Software Tooling	M	191
P11	Senior Director of Technology	26	Data Technologies	M	198
P12	R&D Director	27	Enterprise Software	L	205
P13	Senior Software Engineer	14	Software Tooling	M	215
P14	Senior R&D Manager	16	Enterprise Software	L	220
P15	Chief Software Architect	11	Cloud Services	M	228
P16	Chief Technology Officer	12	Consultancy	S	231
P17	Co-Founder	33	Consultancy	XS	234
P18	Founder	22	Mobile Applications	XS	235

ID: participant identifier; Role: current participant role; Ex: industrial experience (years); OS: organization size (XS<20; S<100; M<500; L<5K; XL>10K); CC: Cumulative number of codes per participant.

and by following theoretical sampling [12], the other interview questions emerged iteratively. This strategy, following GT principles, is meant to let participants express their main concerns on ATD in their own words, and the researcher to explore unsaturated concepts. In addition, we also gathered data on the professional background of participants via a predefined set of demographic questions to collect the data summarized in Table 1.

Interviews were audio-recorded and transcribed manually by following the denaturalism approach, that is, grammar was corrected, interview noise (e.g., stutters) was removed, and nonstandard accents (i.e., non-majority) were standardized, while ensuring a full and faithful transcription [25].

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 [9]. The values reported in column "CC" of Table 1, display the slow increase of cumulative unique codes w.r.t. the number of participants, indicating that we achieved saturation around P16.

2.2 Data Analysis

We followed Glaser's grounded theory data analysis and synthesis processes to create our theory: open coding, selective coding, and theoretical coding [12] [9]. Specifically we examined the whole body of text transcripts, subdivided them into separate "incidents" (sentences or paragraphs) [12], and labeled the incidents with codes to let the theory concepts emerge. When possible, codes are generated by directly quoting the incidents (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 core descriptive categories, which guided the future data collection. Finally, we established the conceptual relations between the different emerging core 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 Figure 1).

Numerous concepts of our theory possess a multifaceted nature. For instance, the concept of "technical debt" itself can be both a *cause*, leading to the introduction of additional debt, and a *consequence*, e.g., of pre-existing debt. Following GT principles, concepts with multiple facets were coded according to the one deemed most important by participants. This ensured the emergence of concepts from the data, rather than from preconceived knowledge of the authors.

During the entirety of the coding procedures, we made use of *memoing* [12]. We created textual memos to elaborate concepts (i) related to single incidents, such as *"This incident exemplifies the impossibility to implement new functionality due to ATD"* and (ii) orthogonal to multiple incidents (e.g., relations between concepts, or categories, such as *"Developer's intuition can lead both to ATD identification and prioritization"*).

As described in Section 2.1, we analysed our data immediately and continuously, using simultaneous data collection and analysis, guided by theoretical sampling. Additionally, during data analysis, we constantly compared our data, memos, codes, and categories, in order to identify and keep track of common notions, topics, and patterns, as they emerged. Similarly, we continuously sorted our memos to evolve the emerging concepts and categories to best fit our codes, leading to the formulation of a substantive, cohesive theory. We performed continuous comparison until additional data being collected did not add new knowledge about the categories, i.e., until we reached the state of saturation (see Section 2.1).

Three researchers were involved in both the data collection and analysis phases, where the first author carried out the coding, memoing, and analysis processes, while the others collaboratively analysed and reviewed iteratively the results.

3 Results

An overview of our grounded theory on ATD is depicted in Figure 1. In this section we describe the 6 core categories emerging from our data, which constitute the foundation of our grounded theory on ATD[3]. We also discuss the emerging relations between the different categories. In line with the grounded theory literature, this enables us to both present comprehensively the emerging theory, and offer explanations and predictions underlying ATD related phenomena [9].

At the core of our theory lies the ***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 also observed from the numerous relations between this category and the other ones reported in Figure 1.

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*

[3] Due to space limitations, in this paper we do not discuss in detail the categories with direct semantics in our theory (*ATD*, *Artifact*, *Tool*, and *System*), and the marginal categories related to human factors (*Person* and *Communication*).

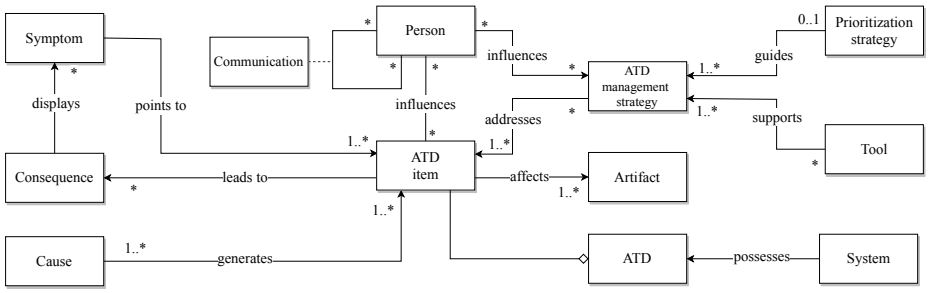

Fig. 1. Core categories of the ATD theory and their relations

one thing, but the plan is 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 [R-Q1]

As causes can generate one or more ATD items, so ATD items can lead to one or more ***consequences***, e.g., reduced development velocity, higher maintenance cost, impossibility to implement new functionality (see Section 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-Q2]

Consequences can *display* one or more ***symptoms***, e.g., recurrent customer, performance, and/or development issues. A consequence can also not display any symptom, either because an 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 [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, multiple 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 [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 [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 [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 [15] (see Section 3.6). 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 technical 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. And if something was really painful, we would know."* -P9 [R-Q7]

ATD management strategies can also be supported by ***tools***, e.g., static analyzers and linters. In some rare instances tools for detecting architectural problems, like component dependency anti-patterns, are used. Nevertheless, in most of the cases, ATD management strategies are not supported by any tools, potentially due to their perceived immaturity: *"The really expensive type of debt [ATD], I have not seen a tool which is able to detect that..."*-P10 [R-Q8]

Two marginal categories emerging in our theory are *person* and *communication*. Being related to *human factors* [5], the nature of these categories is different from the others. The relation between **person** and ATD items is of a multi-faceted nature: among others, people's personal drive, skill set, and awareness can influence ATD from its establishment to its prioritization, and resolution. Further, ATD in a software system often leads to **communication** of ATD-related concepts among the people working on the system. ATD communication may regard the exposition of ATD items, the impediments related to discussing ATD, and even uneasy discussions on who is to blame.

As *person* and *communication* categories emerge as subsidiary categories in our theory, we focus the description of our theory on the categories related the closest to our core category (i.e. *ATD item*). Nevertheless, for the sake of completeness, a discussion of the *person* and *communication* categories is reported in the companion material of this study[4].

3.1 ATD Items

In this section we present the five most prominent types of ATD items residing in software-intensive systems which emerged from our results.

The Minimum Viable Product (MVP) that stuck. Often ATD manifests itself in a software-intensive system as 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

[4] `http://s2group.cs.vu.nl/files/ATD_GT_ECSA_companion_material.pdf`

an immature artifact as architectural basis. This ATD item 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 for quite a long time, we just kept adding new functionality, and this problem was never solved."*-P6 [ATDI-Q1]

The Workaround that stayed. ATD can be introduced in a software system as a temporary workaround 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 [ATDI-Q2]

Consolidated workarounds can become so embedded into an architecture that, while their consequences is 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 [ATDI-Q3]

Re-inventing the Wheel. This type of 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 [ATDI-Q4]

In addition to the resources required to implement already available solutions, drawbacks include lower quality, additional maintenance, 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 [ATDI-Q5]

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. Individual 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 [ATDI-Q6]

Source Code ATD. These are technical debt items strictly related to the implementation of architectural components, and the relations between them. As described by P13: *"It was not really clear what was common and what was separated between the modules ..."*-P13 [ATDI-Q8]

This type of debt is often associated with poor separation of concerns, and/or tightly coupled architectural components, lowering the overall software quality, and directly affecting maintainability, modifiability, and adaptability: *"For example [consider] GDPR. They changed their policy, but our change was harder because our code was just one big clog. Either we built on top of it, making everything even harder, or we separated the pieces"*-P8 [ATDI-Q9]

As further discussed in Section 3.5, this type of debt can be very expensive to fix, and can even originate from systematic processes aimed at lowering ATD: *"We did a rewrite, and there the tight coupling started"*-P13 [ATDI-Q10]

Architectural Lock-in. Related to the previous debt item, ATD can arise in architectural components which, due to their deep embedment into a software architecture, 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 [ATDI-Q11]

New Context, Old Architecture. The last type of ATD item that emerged in our theory regards not paying continuous effort in order to keep the architecture of a software-intensive system aligned with its context, leading to an outdated architecture. This item is mostly incurred 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 [ATDI-Q13]

3.2 Causes

In this section we discuss the four lead root causes of ATD items emerging from the data gathered for our theory.

Time Pressure. Sixteen of the eighteen participants acknowledged time pressure as the leading cause of ATD. P11 summarized: *"In a product you need to hit quarterly targets. Always on the treadmill, getting things done."*-P11 [CA-Q1]

As [R-Q1] evinces, under time pressure, architectural quality is often sacrificed. This is a recurrent theme across participants. P2 noted: *"When time becomes tight, the first thing falling out is cleaning the architecture."*-P2 [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. As P13 stated: *"One thing is always time, it's quicker to do feature development instead of doing architectural changes"*,-P13 [CA-Q4]

From our data emerges that developers often accumulate ATD when dealing with time pressure, under the (often incorrect) assumption that these shortcomings will be dealt with at a later stage, as further detailed in Section 3.6.

Lack of Architectural Knowledge and Documentation. 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 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. So you are constantly extinguishing this little fires to figure out what is going on, it takes a while . . . "*-P12 [CA-Q5]

In addition to the introduction of ATD, lack of architectural knowledge can also lead to the obfuscation of ATD items, hindering the awareness of the ATD

present in a software system. P2 described: *"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 [CA-Q6]

Unsuitable Architectural Decision. ATD can arise by making inadvertently an inappropriate architectural decision. Often, inadvertent design decisions leading to ATD are associated to the lack of context awareness, resulting 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 the options."*-P14 [CA-Q7]

The magnitude of the ATD associated to unfitted decisions varied greatly across participants, with some notable cases where the impact on a product was enormous: *"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 [CA-Q8]

Human influence. Lastly, a recurrent concept of ATD cause regards the influence of human factors on ATD. Under this category fall aspects related to personal drive, such as the example reported in [ATDI-Q7], including lack of developer expertise and cognitive biases (notably the Dunning-Kruger effect [17]).

3.3 Consequences

In this section we document the 4 most prominent consequences of ATD which emerged from our data.

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 in this way."*-P1 [CO-Q1]

To mitigate the negative impact of the carrying on customer perception, some participants reported to actively invest resources to make refactoring efforts tangible to 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 [CO-Q3]

Implementing new functionality becomes challenging. Associated to the carrying cost, ATD can also affect the ease with which new functionalities are implemented. This is often associated with "blurred" responsibilities among architectural components (cf. [ATDI-Q8]). In some cases, due to ATD, it can become necessary to completely discard functionality implementation. Especially telling are instances where such functionalities are characterized by a supposedly trivial implementation. P6 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 [CO-Q5]*

In the most severe cases, architectures can become "crystallized", i.e., ATD hinders almost completely the implementation of new functionalities. One of this rare cases was described by P4: *"They [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"-P4 [CO-Q6]

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 [CO-Q7]

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. This is often occurring in the presence of overloaded components, i.e., components encapsulating a big portion of the business logic or data of a software intensive-system. P14 describes one of such incidents as follows: *"The module became very popular, we just kept building features on it . . . and now it's a bottleneck, because we have many teams working on it at the same time, people are stepping on one another toes."*-P14 [CO-Q8]

3.4 Symptoms

Four types of symptoms, pointing to ATD items, emerged in our theory.

Recurrent Customer Issues. Among all symptoms of ATD, recurring customer issues is the most apparent. 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 [S-Q1].

With this symptom are often associated recurrent patches in the same area of the code, pointing to an architectural problem, P9 describes: *"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 after a while, you realize that you're kind of like. . . you're playing whack-a-mole! It can't be that everything is an edge case!"*-P9 [S-Q2]

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. P10 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."*-S13 [S-Q3]

Performance issues. Performance issues which are hard to address can also be a symptom of ATD. Commonly, performance issues caused by ATD are either *scalability issues*, representing the inability of systems to scale due to ATD, or *performance stalls*, i.e., performance bottleneck 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 [S-Q4]

"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! It might do its job well, but no one wants to touch it!"*-P12 [S-Q6]

3.5 Management Strategies

Six managements strategies to cope with ATD emerged from our data. We identified three types of management strategies, namely *active*, *reactive*, and *passive*.

Active management strategies. Active strategies are based on the acknowledgment of the presence of ATD in a software system, and the development of a plan to actively manage it. In the following we present the 3 active management strategies emerging in our theory.

Boy scout rule. This management strategy borrows from the camping rule "Always leave the campground cleaner than you found it". Based on this metaphor, developers pay back the debt in small incremental steps while carrying out other development activities on a software component, such as new functionality implementation or bug fixes. 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 [MS-Q1]
This strategy is rarely applied. In fact, unlike other forms of TD, ATD is in most cases hard, or even impossible, to be addressed in small increments.

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. 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 [MS-Q2]

Technical credit. This strategy regards the investment of resources to improve architectural maintainability and evolvability prior to the emergence of ATD. Specifically, this strategy aims at mitigating future ATD by proactively improving architectural elements which could slow down future development. Some participants described this strategy from a theoretical standpoint. Nevertheless, the common agreement is that, due to time pressure and uncertain pay-off, 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? 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 [MS-Q3]

Reactive management strategies. Reactive strategies entail that, while the presence of ATD is acknowledged, its management is postponed until the repayment becomes unavoidable (e.g., when ATD prevents the development of a new feature). Two prominent reactive strategies emerged in our data, namely *opportunistic patching* and *major refactoring*.

Opportunistic patching. This strategy, rather than aiming at resolving ATD, 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-Q2], opportunistic patching rarely resolves the root cause of an ATD item, but can nevertheless point to the underlying problem. P11 described a similar situation: *"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... we were just kicking the can down the road... in retrospective we were just patching, patching all the way."*-P11 [MS-Q4]

Major refactoring. Due to ATD severity, it can become necessary to methodically eradicate it, even at the cost of sacrificing other development activities. This constitutes a major undertaking, causing the loss of competitive advantage, and the investment of a conspicuous amount of resources. This strategy includes refactoring conducted by entire developer teams, or even complete rewrites of a products. Due to the resources required, and its uncertain outcome, timing this strategy is a complex problem. P11 explains: *"You always have to overcome this lump of "when is the right time?". There is never a right time. You have to decide when it is. It [ATD] has to reach a crest before you realize: "OK this is enough now", you bite the bullet, and try to do something about it."*-P11 [MS-Q5]

Passive management strategy. The passive management strategy, rather than aiming to actively or passively resolve ATD, attempts to cope with it by carrying out development activities by avoiding to address ATD items.

Neglect. Participants described strategies in which, while the negative impact of the ATD of a system is evident, the cost of fixing it is 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 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 [MS-Q6]

3.6 Prioritization Strategies

In this section, we discuss our findings related to how the refactoring of ATD items is prioritized w.r.t. other development activities, such as feature development and bug fixes. Prioritization strategies 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 [15], i.e., by making a distinction between functional features, bug fixes, architectural

features, and technical debt. Nevertheless, while ATD items are often traced, prioritizing their refactoring w.r.t. 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 [PR-Q1]

This "gut feeling" has been a recurrent theme among participants on how ATD is 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 (cf. [R-Q7]). P3 further clarifies this concept: *"I would say, find your balance, do the minimum necessary. It is not a science, it's an art. Why do large companies fail? Because at some point that balance is tilted."*-P3 [PR-Q2]

4 Related Work

As recommended by Glaserian GT principles [12], 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 et al. [23] 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 [23]). Regarding the results, [23] presents a taxonomy of ATD items and a model of their effects: the specific *ATD items* 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*. Further, a previous work of the same authors [24] 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 [23,24] and our substantive theory, with the ultimate goal of formulating a formal theory [29] of ATD.

Besker et al. [4] 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 [4], such as *ATD detection*, *ATD identification*, *ATD measurement*, *ATD monitoring* and related concepts, did not emerge in our theory. Rather than attributing the absence of such concepts to unsaturation, we conjecture that such divergence in results is due to the research methodology followed. In fact, we can observe that the missing concepts are related to ATD aspects which, while actively discussed in academic settings (e.g. *ATD identification* [31]), 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.

Li et al. [19] present a set of architectural viewpoints and related metamodel for documenting ATD. The viewpoints were constructed via an iterative process driven by the stakeholder concerns on ATD. The viewpoint metamodel partially overlaps with some categories of our theory. However, by focusing on documenting ATD, it aims at the exhaustive characterization of ATD items. Differently, our theory shifts the focus from documentation of ATD items specifics, to the phenomena surrounding them, and as such, it is more encompassing, yet less detailed.

A broader review of the literature shows that the most studied type of technical debt is *source-code ATD* [31] [18], such as ATD related to component dependency [26] or modularity [20]. This typology of ATD emerged in our theory as a specific concept of the *ATD Item* category, namely *source-code ATD*. This category is also mentioned in Brooks's popular book "The Mythical Man Month" [6], 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" [7], 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 [28], 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 "compliance violations" have been studied exclusively in narrower pockets of research [31] [18] [21], and are mapped to our category "new context, old architecture". In [22], 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 3.6). The literature further investigates other emerging categories, such as TD management strategies [2], and the impact of TD on morale [8], but does not systematically focus on the architectural level as we do.

Thanks to the adoption of the Glaserian GT method [12], our theory emerged independently from prior theories and, as such, either confirms or adds to them. This may pave the way for future works toward a joint formal theory [29].

5 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, as customary in grounded theory (e.g., [14]), the verifiability of our results should derive from the soundness of the research method followed. Therefore, we report in Section 2 an in-depth description of the method followed, and (within space constraints) we reference as much as possible to direct quotes from our participants (albeit excerpted).

Our report demonstrates how the emerging theory fulfills the grounded theory evaluation criteria [9], specifically: (i) our categories *fit* the underlying data, (ii) the theory is able to *work* (i.e., explain ATD related phenomena), (iii) the theory

has *relevance* to the domain (i.e., development practices of large and long-lived systems), and (iv) the theory is *modifiable* as new data appears.

As any grounded theory study, our investigation establishes a mid-range substantive theory, i.e., a theory where elements belonging to the studied context can be transferred to other contexts with similar characteristics. We hence do not claim our theory to be absolute or final, and we highly welcome its extension, e.g., by refining its granularity and adding detail to emerging concepts, or even unveiling new concepts and categories that did not emerge in this investigation.

6 Conclusions

Our investigation presents structured insights into the challenges faced in industrial settings when dealing with ATD. From our study emerged a set of interrelated categories regarding ATD, leading to a cohesive theory of ATD that connects its causes, consequences, symptoms, management strategies, and other related phenomena. We made a deep-dive into each category, by grounding our findings in the experience of knowledgeable software practitioners. Our theory provides a solid empirical foundation which may benefit both (i) *practitioners* aiming at a better understanding of the ATD they experience, and (ii) *researchers* looking for a theoretical framework of how ATD is experienced in industrial settings. Notably, among other results, from our investigation emerge a set of symptoms, consequences, and management strategies on which future research, methodologies, and tooling, can be based. 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 [30].

References

1. Adolph, S., Hall, W., Kruchten, P.: Using grounded theory to study the experience of software development. Empirical Software Engineering **16**(4), 487–513 (2011)
2. Alves, N., Mendes, T.S., de Mendonça, M.G., Spínola, R.O., Shull, F., Seaman, C.: Identification and management of technical debt: A systematic mapping study. Information and Software Technology **70**, 100–121 (2016)
3. Avgeriou, P., Kruchten, P., Ozkaya, I., Seaman, C.: Managing Technical Debt in Software Engineering. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2016)
4. Besker, T., Martini, A., Bosch, J.: Managing architectural technical debt: A unified model and systematic literature review. Journal of Systems and Software **135** (2018)
5. Bourque, P., Fairley, R.E., IEEE Computer Society: Guide to the software engineering body of knowledge (2014)
6. Brooks Jr, F.P.: The mythical man-month (anniversary ed.), Addison-Wesley (1995)
7. Fowler, M.: Refactoring: improving the design of existing code. Addison-Wesley Professional (2018)
8. Ghanbari, H., Besker, T., Martini, A., Bosch, J.: Looking for peace of mind?: manage your (technical) debt: an exploratory field study. In: ACM/IEEE EMSE Symposium (2017)
9. Glaser, B.: Theoretical Sensitivity. Sociology Press (1978)
10. Glaser, B.: Basics of grounded theory analysis: Emergence vs forcing. Sociology press (1992)

11. Glaser, B.: The Grounded Theory Perspective III: Theoretical Coding. Sociology Press (2005)
12. Glaser, B., Strauss, A.: Discovery of grounded theory: Strategies for qualitative research. Aldine (1967)
13. Glaser, B.G., Holton, J.: Remodeling grounded theory. In: Forum Qualitative Sozialforschung/Forum: Qualitative Social Research. vol. 5 (2004)
14. Hoda, R., Noble, J.: Becoming agile: a grounded theory of agile transitions in practice. In: International Conference on Software Engineering. IEEE Press (2017)
15. Kruchten, P.: What Colour Is Your Backlog? (2008), available Online: `https://tinyurl.com/y6f7vhpx` (Accessed 10th May 2020)
16. Kruchten, P., Nord, R., Ozkaya, I.: Technical debt: from metaphor to theory and practice. IEEE Software **29**(6), 18–21 (2012)
17. Kruger, J., Dunning, D.: Unskilled and unaware of it: how difficulties in recognizing one's own incompetence lead to inflated self-assessments. Journal of personality and social psychology **77**(6), 1121 (1999)
18. Li, Z., Avgeriou, P., Liang, P.: A systematic mapping study on technical debt and its management. Journal of Systems and Software **101**, 193–220 (2015)
19. Li, Z., Liang, P., Avgeriou, P.: Architecture viewpoints for documenting architectural technical debt. In: Software Quality Assurance, pp. 85–132. Elsevier (2016)
20. Li, Z., Liang, P., Avgeriou, P., Guelfi, N., Ampatzoglou, A.: An empirical investigation of modularity metrics for indicating architectural technical debt. In: International ACM Conference on Quality of software architectures (2014)
21. Martini, A., Bosch, J.: The danger of architectural technical debt: Contagious debt and vicious circles. In: WICSA Conference. IEEE (2015)
22. Martini, A., Bosch, J.: Towards prioritizing architecture technical debt: information needs of architects and product owners. In: Euromicro Conference on Software Engineering and Advanced Applications. pp. 422–429. IEEE (2015)
23. Martini, A., Bosch, J.: On the interest of architectural technical debt: uncovering the contagious debt phenomenon. Journal of Software: Evolution and Process (2017)
24. Martini, A., Bosch, J., Chaudron, M.: Investigating architectural technical debt accumulation and refactoring over time: A multiple-case study. Information and Software Technology **67**, 237–253 (2015)
25. Oliver, D., Serovich, J., Mason, T.: Constraints and opportunities with interview transcription: Towards reflection in qualitative research. Social forces p. 1273 (2005)
26. Roveda, R., Fontana, F.A., Pigazzini, I., Zanoni, M.: Towards an architectural debt index. In: Euromicro Conference on Software Engineering and Advanced Applications. IEEE (2018)
27. Stol, K.J., Ralph, P., Fitzgerald, B.: Grounded theory in software engineering research: a critical review and guidelines. In: IEEE/ACM International Conference on Software Engineering (2016)
28. Szyperski, C., Gruntz, D., Murer, S.: Component software: beyond object-oriented programming. Pearson Education (2002)
29. Urquhart, C., Lehmann, H., Myers, M.D.: Putting the theory back into grounded theory: guidelines for grounded theory studies in information systems. Information systems journal **20**(4), 357–381 (2010)
30. Verdecchia, R., Lago, P., Malavolta, I., Ozkaya, I.: ATDx: Building an Architectural Technical Debt Index. In: ENASE Conference (2020)
31. Verdecchia, R., Malavolta, I., Lago, P.: Architectural Technical Debt Identification: the Research Landscape. In: IEEE/ACM TechDebt Conference (2018)