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

Citation for the original published paper (version of record):

Verdecchia, R., Kruchten, P., Lago, P. et al (2021). Summary: Building and evaluating a theory of architectural technical debt in software-intensive systems. CEUR Workshop Proceedings, 2978

Summary: Building and evaluating a theory of architectural technical debt in software-intensive systems*

Roberto Verdecchia[1], Philippe Kruchten[2], Patricia Lago[1,3] and Ivano Malavolta[1]

[1]Vrije Universiteit Amsterdam, The Netherlands
[2]University of British Columbia, Vancouver, Canada
[3]Chalmers University of Technology, Gothenburg, Sweden

Abstract

This paper reports a summary of a study on Architectural Technical Debt (ATD) published in the Journal of Software and Systems [1]. By borrowing from the 16162 definition of technical debt, we can define ATD as a collection of design or implementation constructs, present at the architectural level of software-intensive systems, that are expedient in the short term, but set up a technical context that can make future changes more costly or impossible. In the study we aimed at investigating how software practitioners conceptualize ATD, and how they deal with it. In order to do so, we conducted a mixed-method empirical study constituted by a Glaserian grounded theory, followed by an evaluation and refinement of the emerging theory *via* focus groups. The result of our investigation constitutes an encompassing conceptual model of architectural technical debt, identifying and relating concepts such as its symptoms, causes, consequences, management strategies, and communication problems. The emerging theory can support both research and practitioners with structured knowledge about the crucial factors of architectural technical debt experienced in industrial contexts.

Keywords

Software Engineering, Software Architecture, Technical Debt, Software Evolution, Grounded Theory, Focus Group

1. A Theory of Architectural Technical Debt

In this paper, we provide a high-level overview of our theory on Architectural Technical Debt, which is documented in its entirety in the journal publication *"Building and evaluating a theory of architectural technical debt in software intensive systems"* [1]. The study leveraged a mixed-method empirical study, constituted by a Glaserian grounded theory, followed by an evaluation

*Use the original publication when citing this work. R. Verdecchia, P. Kruchten, P. Lago, I. Malavolta, *Building and evaluating a theory of architectural technical debt in software-intensive systems*, Journal of Systems and Software. 176 (2021). doi: https://doi.org/10.1016/j.jss.2021.110925.

ECSA'21: European Conference on Software Architecture, September 12–17, 2021, Virtual

r.verdecchia@vu.nl (R. Verdecchia); pbk@ece.ubc.ca (P. Kruchten); p.lago@vu.nl (P. Lago); i.malavolta@vu.nl (I. Malavolta)
https://robertoverdecchia.github.io/ (R. Verdecchia); https://philippe.kruchten.com/ (P. Kruchten); http://patricialago.nl/ (P. Lago); http://www.ivanomalavolta.com/ (I. Malavolta)
0000-0001-9206-6637 (R. Verdecchia); 0000-0003-1359-4867 (P. Kruchten); 0000-0002-2234-0845 (P. Lago); 0000-0001-5773-8346 (I. Malavolta)

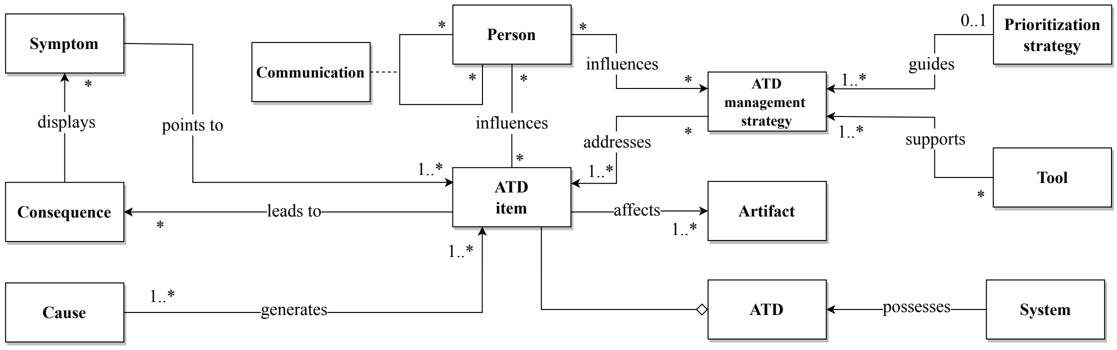

Figure 1: Core categories of the ATD theory and their relations [3]

and refinement of the emerging theory *via* focus groups.

In the following, we provide a concise summary of the categories emerging from our study, which constitute the foundation of our grounded theory on architectural technical debt. In addition, we provide some hints on the types and concepts of our theory, which constitute the finer-grained building blocks of the theory presented in our work [1]. The complete study [1] describes in detail each category, type, and concept of our theory, supported by extensive examples on how these manifest themselves in practice as described by industrial practitioners. A high-level overview of our theory on architectural technical debt is depicted in Figure 1.

System: The *system* category represents the system being developed. In our 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" [2]. A system possesses a certain amount of architectural technical debt, that usually accumulates in time, and is managed *via* different *management strategies.*

Architectural Technical Debt (ATD): The *ATD* category represent the entirety of the technical debt incurred at the architectural level in a software-intensive system. In other words, ATD embodies 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.

ATD Item: At the core of our theory lies *ATD item*, *i.e.,* the category that represents the instances of ATD residing in a software-intensive system. ATD items belong in our theory to one of three mutually exclusive types, namely *framework ATD items*, *process ATD items*, and *implementation ATD items.* Framework ATD items are specific to the adoption and adaptation of software frameworks in software projects, and entail for example the use of unfitted, superfluous, or not updated frameworks. Process ATD items instead regard the high-level processes of architecting and managing software-intensive systems, such as neglected architecture maintenance and evolution, or minimum-viable products used as immature architectural foundation for the development of a software-intensive system. Finally, implementation ATD items focus on lower-level implementation details that, due to their widespread impact on the maintenance and evolution of a software-system, become of architectural relevance. Examples of implementation

ATD include segments of code affected by technical debt, or ill-designed components created *via* iterative trial-and-error development activities.

Cause: At the root of each ATD item lies one or more *cause*. Each cause can generate one or more ATD items. Causes belong in our theory to two different types, namely *internal* and *external causes*. Internal causes embody factors inherent to the development and maintenance of the system, while external causes regard the influence of the context of software-intensive systems on their ATD. Prominent examples of internal causes are the lack of architectural knowledge, unsuitable architectural decisions, lack of anticipation, and human factors. Notable external causes are instead the simple passing of time, time pressure, business pressure, and the misalignment of an architecture with its context.

Consequences: As causes can generate one or more ATD items, so ATD items can lead to one or more *consequences*. From our investigation, three different types of consequences emerged, namely *business-related consequences*, *functionality-related consequences*, and *product-development-related consequences*. Business-related consequences regard financial aspects of software development, such as a carrying cost spent for maintenance and evolution of a system, the loss of business opportunities, or a higher risk to incurring in other ATD consequences. Functionality-related consequences instead affect working on functionalities of a software-intensive system, and range from impediments arising when maintaining a functionality, to the impossibility to implement new functionalities due to a completely crystallized architecture. Finally, product-development-related consequences affect development activities, and manifest themselves as difficulties in carrying out parallel work, or even a persistent flakiness that makes the behaviour of a software-intensive system unpredictable. Interestingly, ATD items can also be "dormant", *i.e.,* the ATD items are present in the system, but do not lead to any immediate consequence.

Symptoms: Consequences can display *symptoms*. Rather than being ATD items *per se*, similar to the medical domain, symptoms can point to the presence of ATD items in a software-intensive systems, especially if more symptoms co-occur. Symptoms can indicate the presence of one or more ATD items, *i.e.,* observing symptoms displayed by a consequence can lead to the identification of one or more *ATD items*. In the most recurrent cases, a multitude of symptoms point to a single, widespread, ATD item. In our study we identified four types of symptoms, namely symptoms related to *issues*, *resources*, *performance*, and *development practices*. ATD symptoms related to issues can emerge as recurrent costumer issues, recurrent patches, a high number of defects, or even security breaches. Resource-related symptoms instead can manifest as growing maintenance activities, the need of senior or specialized staff, or growing monetary resources needed to keep a software-intensive system running. Regarding issues related to performance, these can appear either as scalability issues, or performance stalls that cannot be resolved without a major architectural refactoring. Finally, development-related symptoms emerged in our study as the instinctual refrain of software developers to modify a certain component where ATD resides, or as functionalities implemented in the wrong component, or as the appearance of multiple yet inconsistent instances of the same data throughout a software-intensive system. Interestingly, ATD items can also not display any symptom, either because the ATD items are "dormant", or because the observed symptoms are not sufficiently distinct to establish the relation.

ATD Management Strategies: ATD items can be addressed via one or more *ATD management strategies*. Similarly, it is possible to address multiple ATD items with a single management strategy (typically *via* rewrites). In our work, we identified three types of management strategies, namely *active*, *reactive*, and *passive management strategies*. Active strategies are based on the acknowledgment of the presence of ATD, and the development of a plan to actively manage it. Active management strategies include the application of the "boy scout rule", systematically dedicating time to address ATD, and building up technical credit to mitigate the impact of the ATD that could emerge in the future. Reactive management strategies entail the postponement of refactoring activities until the repayment becomes unavoidable, and range from superficial opportunistic patches, to major refactoring activities, and even the rewrite from scratch of entire software-intensive systems due to "technical debt bankruptcy".

Tools: ATD management strategies can be supported by *tools*, *e.g.,* static analyzers and linters, such as Clang Tidy[1] and SonarQube[2]. In our investigation, the adoption of tools to explicitly identify and manage ATD did not emerge as an established industrial practice, possibly due to the perceived immaturity or usefulness of existing ATD-centric tools.

Artifacts: ATD items can affect and reside in one or more *artifacts*, *e.g.,* architectural components, test suites, and documentation. In addition, ATD can also occur in the relation established between two or more of these artifacts. Commonly, given the widespread nature of ATD items, numerous artifacts are simultaneously affected by a single ATD item. While architectural components are the ever-present artifacts in which ATD items manifests themselves, test suites result also to be often affected by ATD items residing in architectural components. Similarly, ATD items can be reflected in a partial, absent, or even erroneous documentation of the architecture of a software intensive-system.

Prioritization strategy: 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 [4]. ATD prioritization strategies can leverage one or more management strategies, depending on the specific context considered and the ATD item(s) regarded. Due to difficulties with quantifying the impact of ATD, practitioners do not adopt systematic prioritization approaches for ATD; rather, they use informal ones, such as a personal "gut feeling", to balance the refactoring of their ATD with other development activities.

Person: 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. People can highly influence ATD items, from the establishment of ATD items to their resolution. Similarly, also ATD items can influence people, as the encompassing and complex nature of ATD can affect development activities over a prolonged period of time, leading to severe consequences on the morale of developers. Awareness of people also plays a role in ATD, as to be able to manage ATD, one must first be aware of its presence. Personal drive, seniority, and skill set of people are also related to ATD, as these can lead to the championing of refactoring ATD items, or the lingering of an ATD item in a software-intensive system for a long time. In addition, the intuition of people can lead to the identification, prioritization and management of ATD, while

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

numerous cognitive biases can instead lead to the inadvertent introduction new ATD items.

Communication: ATD can lead to the *communication* of concepts related to it among people working on a software-intensive system where the ATD resides. More specifically, people can explain ATD, *i.e.,* rise awareness among developers, managers, and the like, of the presence of ATD items. Rising awareness results to be an important aspect steering ATD management and prioritization strategies. ATD can also lead to communication impediments, as rising awareness on the severeness of the ATD present in a software-intensive system is not always an easy task. In the most problematic cases, ATD can lead to friction among people working on a software-intensive system, resulting in blaming people who potentially incurred in ATD items, or did not manage them properly. Finally, in our theory emerged difficulties in communicating the presence of ATD to the stakeholders of software intensive-systems. In fact, as stakeholders might not possess the same technical insights as developers, as the ATD in a software-system grows, so do the difficulties in explaining why time needs to be allocated to refactoring activities instead of implementing new functionalities.

References

[1] R. Verdecchia, P. Kruchten, P. Lago, I. Malavolta, Building and evaluating a theory of architectural technical debt in software-intensive systems, Journal of Systems and Software 176 (2021). doi:https://doi.org/10.1016/j.jss.2021.110925.

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

[3] R. Verdecchia, P. Kruchten, P. Lago, Architectural Technical Debt: A Grounded Theory, in: European Conference on Software Architecture, Springer, 2020, pp. 202–219. doi:10.1007/978-3-030-58923-3_14.

[4] P. Kruchten, What Colour Is Your Backlog?, 2008. Available Online: https://tinyurl.com/y6f7vhpx (Accessed 10th May 2020).