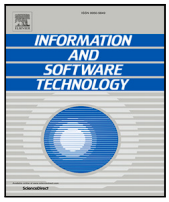
Information and Software Technology 139 (2021) 106669

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



Architectural design decisions that incur technical debt — An industrial case study

Mohamed Soliman *, Paris Avgeriou *, Yikun Li

Bernoulli Institute for Mathematics, Computer Science and Artificial Intelligence, University of Groningen, Groningen, The Netherlands

ARTICLE INFO

Keywords:
Technical debt
Architectural design decisions
Architectural knowledge
Architectural technical debt

ABSTRACT

Context: During software development, some architectural design decisions incur technical debt, either deliberately or inadvertently. These have serious impact on the quality of a software system, and can cost significant time and effort to be changed. While current research efforts have explored general concepts of architectural design decisions and technical debt separately, debt-incurring architectural design decisions have not been specifically explored in practice.

Objective: In this case study, we explore debt-incurring architectural design decisions (DADDs) in practice. Specifically, we explore the main types of DADDs, why and how they are incurred in a software system, and how practitioners deal with these types of design decisions.

Method: We performed interviews and a focus group with practitioners working in embedded and enterprise software companies, discussing their concrete experience with such architectural design decisions.

Results: We provide the following contributions: 1) A categorization for the types of DADDs, which extend a current ontology on architectural design decisions. 2) A process on how deliberate DADDs are made in practice. 3) A conceptual model which shows the relationships between the causes and triggers of inadvertent DADDs. 4) The main factors that influence the way of dealing with DADDs.

Conclusion: The results can support the development of new approaches and tools for Architecture Technical Debt management from the perspective of Design Decisions. Moreover, they support future research to capture architecture knowledge related to DADDs.

1. Introduction

Architectural design decisions (ADDs) have the biggest impact on the quality of a software system, and they are hard to change after their implementation [1]. Some ADDs incur technical debt, i.e. they "set up a technical context that can make future changes more costly or impossible" [2]. We call these *Debt-incurring Architectural Design Decisions (DADDs)*, and their impact is well recognized by both practitioners and researchers [3,4]. DADDs can be either deliberate or inadvertent [5].

Deliberate DADDs are taken because of time pressure or lack of resources: a solution is chosen that is quicker and cheaper but compromises maintainability and evolvability. For example, instead of adhering to the layered structure of the architecture, shortcuts are created that bypass layers. This results in implementing the required features quicker, but those shortcuts create ripple effects when making changes.

Inadvertent DADDs, are decisions that, when taken, do not bear any negative consequences on the system maintainability. However, in the course of time, the decisions cause the development team to spend extra effort on maintenance, thus becoming technical debt items. A typical example of inadvertent DADDs, is when a technology is selected that becomes obsolete after a few years [6]. That technology may have been an optimal decision in the past, but it now causes a lot of workarounds and unnecessary complexity.

Related research work on architectural technical debt (ATD) [7] has empirically explored different types of ATD items (e.g. dependency violations) [8], their causes, trends [9] and effects [8]. Moreover, methods were proposed to identify ATD (e.g. through capturing architectural bad smells from existing systems) [10]. Nevertheless, current studies have not examined ATD from the perspective of the Architecture Design Decisions (ADDs) that incur it either deliberately or inadvertently. This perspective is of paramount importance to inform the development of approaches to manage ATD, as well as tools to support the decision making process.

In this paper, we aim at exploring the current state of practice in industry regarding DADDs: we determine types of DADDs, we study the decision making process behind deliberate DADDs, we explore how and

* Corresponding authors.
E-mail addresses: m.a.m.soliman@rug.nl (M. Soliman), p.avgeriou@rug.nl (P. Avgeriou), yikun.li@rug.nl (Y. Li).

https://doi.org/10.1016/j.infsof.2021.106669
Received 2 September 2020; Received in revised form 4 June 2021; Accepted 5 June 2021
Available online 14 June 2021

why inadvertent DADDs are formed, and finally we investigate how to deal with DADDs after their implementation. To achieve this goal, we performed a case study with multiple high-tech companies working in both embedded systems and enterprise applications, enquiring eleven experienced practitioners in depth, through individual interviews and a joint focus group. We asked practitioners about their concrete experiences with DADDs in their involved projects, and discussed their opinions. In the case study, we limit the scope of technical debt to the internal quality of software systems, as defined by Avgeriou et al. [2]; in other words, we are concerned with the impact of DAADs on maintainability and evolvability.

The case study has resulted in four concrete **contributions**:

- We extend current ADDs classifications [11] with specialized types of decisions that correspond to DADDs.
- We propose a practice-oriented *decision making process* that architects can follow to make deliberate DADDs.
- We identify the main *triggers* that release the occurrence of inadvertent DADDs, as well as their *root causes*.
- We pinpoint the *implications* of implemented DADDs and the *factors* to decide on how to deal with them.

The rest of the paper is structured as follows. Section 2 elaborates on the study design. Sections 3, 4, 5, and 6 present the results on the types of DADDs, deliberate DADDs, inadvertent DADDs and deadling with implemented DADDs respectively. Section 7 discusses the results and their implications. Section 8 reports on the threats to the validity of our study. Section 9 presents related work, while Section 10 concludes the paper and outlines the directions of future work.

2. Study design

2.1. Research questions

To meet the research goal stated in Section 1, we ask the following research questions:

- **(RQ1) What are common types of DADDs?** Researchers have proposed different classifications of ADDs (e.g. [11,12]). Each type of ADD involves selecting an architectural solution; for example "Existence decisions" [11] make concrete changes in the component design. However, DADDs, that bring in a new decision factor (i.e. technical debt) have not been considered by existing classifications of ADDs. Therefore, we ask RQ1 to better understand DADDs, and their relationships to existing classifications of ADDs. Understanding and specifying types of DADDs could support extending current approaches and tools of architectural knowledge management [13] with additional ways to manage DADDs. Consequently, a better way to manage DADDs, could help improving current technical debt management approaches [14].
- **(RQ2) How do practitioners take deliberate DADDs?** Architectural decision making involves stating a design problem, analysing a number of design alternatives, evaluating those alternatives based on specific criteria (e.g. quality attributes) and selecting one alternative with an explicit rationale [15]. Moreover, ADDs are commonly taken based on an agreement between a group of stakeholders; this agreement is quite challenging to reach [7]. Deliberate DADDs represent a special type of ADDs with previously known risks and negative implications on the maintainability and evolvability of a software system. We ask RQ2 to determine how exactly the decision making process of deliberate DADDs takes place. This can help to provide support to software architects and engineers, specifically for making DADDs.
- **(RQ3) Why and how do architecture design decisions become inadvertent DADDs?** Inadvertent DADDs are discovered only after their implementation, and after the developers start paying technical debt. We ask RQ3 to determine the reasons that cause the occurrence of inadvertent DADDs, as well as the factors that trigger them. This might include deficiencies in existing practices of architectural decision making or changes in context. This is useful for practitioners to consider in order to minimize the occurrence of inadvertent DADDs, as well as to monitor their triggers.
- **(RQ4) How are implemented DADDs dealt with?** Knowing that DADDs (deliberate or inadvertent) exist in a software system requires to take action managing these decisions as well as related decisions in further iterations. On the one hand, keeping track and documenting ADDs (and specially their rationale) is a well-known challenge in literature [16]. On the other hand, DADDs present a contingent risk on the quality of the software system. We ask this question to determine best practices for dealing with DADDs in an existing system.

To answer these research questions, we conducted an exploratory case study with multiple practitioners in several organizations. The case and units of analysis are explained in Section 2.2. Data collection and data analysis are described in Sections 2.3 and 2.4 respectively. Fig. 1 shows an overview of data collection and data analysis.

2.2. Cases and units of analysis

The case study in this paper is an embedded, multiple-case study [17], i.e. we study a number of cases, each containing multiple units of analysis. Our cases are high-tech companies, which are specialized in the development of either embedded software or enterprise software systems. The units of analysis are practitioners that have a long, hands-on experience in dealing with DADDs, both deliberate and inadvertent ones. Table 1 shows background information of the practitioners, and the companies where they work; in total, we collected data from eleven experienced practitioners, who work in seven different companies. All practitioners are suitable for the study, in the sense that they take architectural design decisions, and deal with technical debt in their daily work.

To increase the external validity of this study, we consider companies and practitioners in two domains: embedded software and enterprise applications. The two domains face different architectural problems and use different architectural solutions. For instance, embedded software is more interdisciplinary than enterprise applications, while enterprise applications have more technological options compared to embedded software. Thus, selecting practitioners from both domains can minimize conceptual gaps and validate discovered concepts during data analysis (for more details, see the application of grounded theory in see Section 2.4). This aligns with other empirical studies in the field of software architecture (e.g. [18]) that show differences between embedded software and enterprise applications regarding their practices and tools. Similarly to those studies, we selected practitioners using convenience sampling; in fact we followed different ways of convenience sampling to identify practitioners from the two domains, as explained in the following sub-sections.

2.2.1. Subjects from embedded software

We identified five practitioners in a large company, which is specialized in the production of embedded systems. Using a sample of practitioners from a single organization allowed us to answer our research questions in a high level of detail and understanding. This is especially useful in the embedded domain, because the production of embedded software requires strong collaboration among different engineering departments (e.g. mechanical, chemical, and software engineering). Thus, selecting practitioners from the same organization supported our purpose to have more and richer information about the production of embedded systems within the organization. Details about the practitioners' background and the company information are listed in the first five rows of Table 1.

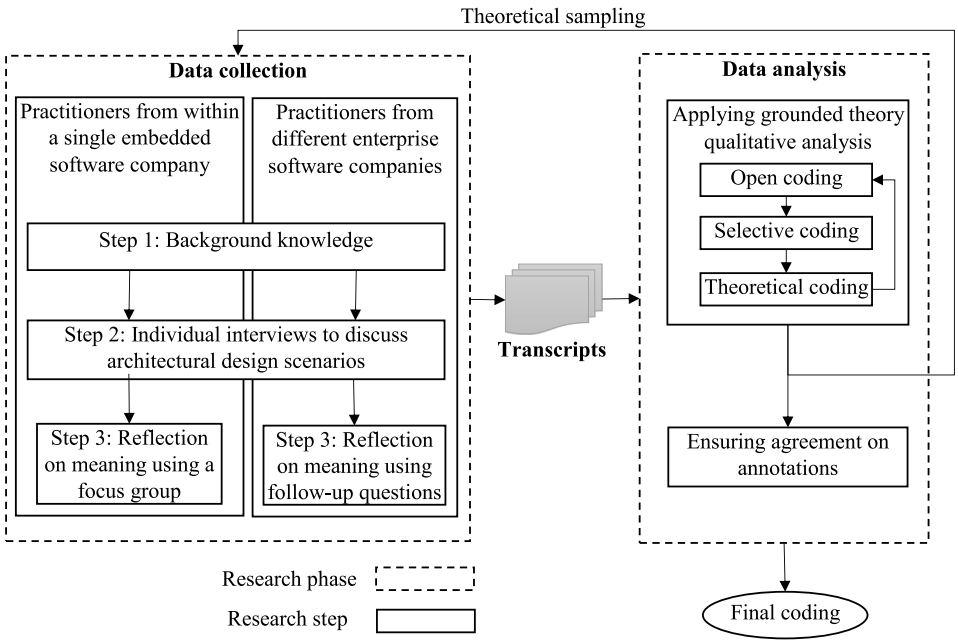


Fig. 1. Data collection and data analysis.

The development of embedded systems in the company is interdisciplinary. Software development in this company also follows agile practices (e.g. a backlog is maintained for new requirements). Moreover, they have systematic methods to apply architectural rules among software engineers in the whole organization. The architectural rules are documented in a reference architecture. Software team members within the organization have specific roles: technical leader, software architect, and software engineer. Technical leaders and software architects are the main design decision makers within the software team.

We selected the practitioners through an agreement with the managers of the company, who facilitated our meetings with the practitioners.

2.2.2. Subjects from enterprise applications

We identified six practitioners, who work in six different companies. Selecting practitioners from multiple organizations allowed us to answer our RQs based on different experiences from multiple companies and in different technological settings. This is especially useful due to the big architectural variety (e.g. using different architectural styles and technologies) between different enterprise applications.

Details about the practitioners' background and their company information are listed in the last six rows from Table 1. All companies are specialized in developing enterprise applications. The identified practitioners work as software architects or technical leaders. Moreover, the six identified interviewees are experienced with dealing with technical debt in their work. We verified this before interviewing them by checking their blog articles on technical debt within their professional LinkedIn[1] profile (4 out of 6 subjects had blog posts related to technical debt). In addition, we validated their experience with technical debt by asking them directly before the interviews.

2.3. Data collection

Our data collection process followed three main steps (left part of Fig. 1) as advised by Seidmann et al. [19]. The steps are explained in the following sub-sections.

[1] www.linkedin.com.

2.3.1. 1st step: Background knowledge

In this step, we asked practitioners to answer questions about their experience, and especially architectural experience. We conducted this step, one week before meeting the practitioners, which allowed us to ensure their suitability for our study. A summary of the practitioners' experiences is presented in Table 1. In addition to background questions, we provided them with a document with basic concepts on software architecture and technical debt to prepare them for the following two steps.

2.3.2. 2nd step: Individual interviews to discuss architectural design scenarios

In this step, we interviewed the eleven practitioners individually and asked them about design scenarios in which debt-incurring ADDs have been taken, both deliberate and inadvertent ones. Interviews provide the best means to capture the experiences of software engineers and the insights they obtained from these experiences [19]. Each individual interview had two main phases:

- *Introductory questions*: We started each interview with some introductory questions (e.g. *What are examples of architectural design decisions? How do you ensure the maintainability of an existing software system?*). The purpose of the introductory questions was to motivate interviewees to speak about their concrete experiences with architectural design decisions and technical debt.
- *Main questions*: In this phase, we asked the interviewees to specifically elaborate on design scenarios about DADDs in detail. We covered both types of design scenarios for DADDs: deliberate and inadvertent. Before asking our questions regarding DADDs, we explained to the interviewees our interest in DADDs, and asked them to focus solely on DADDs and disregard other types of design decisions. For each DADD, we asked the interviewees about their decision making steps (e.g. thinking about design issue and alternative solutions), monitoring, implementation, impact and resolution. Table 2 shows our questions for each DADD.

During the interviews, we allowed the interviewees to speak their mind freely without restricting their answers. We asked also several follow-up questions to steer the conversation when they revealed something of interest to the study. Each interview took on average 45 min. The interviews with the first five practitioners (see Table 1) were conducted

Table 1
Interview participants.

ID	Years of experience		Current role	Location	Company information		
	IT	Architecture			ID	Size	Domain
1	20	15	Technical leader	Netherlands	1	>100,000	Embedded
2	18	12	Software architect	Netherlands	1	>100,000	Embedded
3	30	10	Technical leader	Netherlands	1	>100,000	Embedded
4	16	10	Software architect	Netherlands	1	>100,000	Embedded
5	15	3	Software engineer	Netherlands	1	>100,000	Embedded
6	15	8	Technical leader	USA	2	>100,000	Enterprise
7	20	7	Software architect	Ireland	3	>500	Enterprise
8	28	20	Software architect	USA	4	<50	Enterprise
9	16	8	CIO	USA	5	<50	Enterprise
10	13	7	Software engineer	Belgium	6	<50	Enterprise
11	13	5	Software architect	UAE	7	>100,000	Enterprise

Table 2
Individual interview questions for DADDs.

Concept	Question
Design issue	Which design issues did the decision tackle?
Requirements	Which requirements affect the decision?
Solutions	What were the alternative solutions?
Rationale	Why did you decide on the selected solution?
Group decision	• Was the decision taken in a group? • How did you agree on the solution? • Did software developers agree on this decision?
Implementation	What was the implementation of the solution?
Cause of technical debt (for inadvertent)	Why does this design decision incur technical debt? (This question is needed for deliberate decisions)
Monitoring, tracking	• How did you deal with the debt incurring decision during performing changes in further iterations? • Did you document or keep track of the decision?
Awareness	After the implementation of a debt incurring decision, who was aware of it?
Impact	How did the occurrence of the implemented debt-incurring design decision affect the work of software developers in further iterations?
Resolution	Did you succeed to improve or change the debt-incurring design decision? Why?

face-to-face in a single day. During the first three interviews, both the first and the second authors interviewed practitioners together. This supported an effective interview format with rich follow-up questions and useful answers. The 4th and 5th interviews were conducted in parallel from the 1st and 2nd author separately, but using the same questioning protocol. The rest of the interviews (from the 6th to the 11th interviews) were conducted by the 1st author using the same questioning protocol, and using video conference software. All interviews were recorded and transcribed to be analysed in the next phase (see Section 2.4).

2.3.3. 3rd step: Reflection on meaning

After interviewing each practitioner individually, we communicated with the practitioners again to reflect on our understanding of the design scenarios and ask about their opinions directly (as opposed to asking them about their experiences from the design scenarios during the interviews). The questions were thus different from the individual interview questions. Table 3 shows the questions posed in this step. We communicated with the practitioners using two methods: conducting a focus group and asking follow-up questions, as explained in the following two paragraphs.

Focus group. we met with the first five practitioners (see Table 1) in a plenary session for a focus group. Our main goal from the focus group was to use group dynamics to support exchanging direct opinions between the practitioners. We allowed practitioners to speak and exchange their opinions freely without interruptions. The focus group took 1 h, and was conducted by the first and second authors together. The focus group followed a "dual moderator" format, where the first author ensured that all topics are covered, while the second author ensured that the session progresses smoothly. We recorded and transcribed the discussion within the focus group to facilitate our data analysis (see Section 2.4).

Table 3
Questions for reflection on meaning.

ID	Question
1	What are challenges of making deliberate DADDs?
2	How are stakeholders usually convinced to take DADDs?
3	Do software developers tend to agree on DADDs?
4	How could we better plan DADDs?
5	How could we better deal with DADDs after their implementation?
6	Do you need a process or tool to manage DADDs?
7	Who should be aware about implemented DADDs?
8	What are the reasons of inadvertent DADDs?

Follow-up questions. Since the subjects from the enterprise application domain (practitioners 6 to 11 in Table 1) were distributed in different locations and time zones, it was not possible to conduct a virtual focus group between them. Therefore, to get answers for this step (see Table 3) from these subjects, we sent our questions to them directly. Sending the questions directly made it feasible to get answers from five of the practitioners. We did not get an answer from the 10th practitioner, so for this participant we only included the data from the interview in our data collection. All answers are added to our transcripts for data analysis (see Section 2.4).

2.4. Data analysis

The data analysis process consisted of two main phases, as shown in Fig. 1 and elaborated in the following sub-sections.

2.4.1. Applying grounded theory qualitative analysis

To analyse the transcripts of the interviews, focus group and answers to the follow-up questions, we followed an iterative qualitative data analysis process based on the classical/Glaserian variant of "Grounded Theory" [20–22]. The analysis consists of three steps:

- *Open coding*: We annotated selected statements within the transcripts with new emerging codes (i.e. labels that present certain concepts). Giving the freedom to researchers to decide on new codes during annotations is important in this step to capture new concepts. Beside annotating sentences, we wrote *interesting observations (i.e. memos)* about the codes and their relations. Memos were documented using diagrams to support elaborating codes and identifying gaps. Specifically, we have created two diagrams: one for the deliberate DADDs scenarios and the other for the inadvertent DADDs scenarios. The created memos are available in the replication package.[2]
- *Selective coding*: After annotating all statements with new codes, we *constantly compared* codes and their annotations with each other to merge similar codes together and group related codes into clusters of codes. We did this for both the deliberate and inadvertent DADDs scenarios with the help of the created memos. For example, we annotated a sentence as "time constraint", and another as "availability of resources"; subsequently both codes were assigned to the same group: "deliberate DADDs factors". The replication package[2] contains co-occurrence matrices, which show relationships between the initial codes (after open coding), and the final codes.
- *Theoretical coding*: Groups of codes and their relations were modelled to provide a high-level overview on captured concepts and their relationships. When establishing relationships between concepts, we ensured that these relationships apply to all annotated statements. For example, to answer RQ3, we proposed a model in Fig. 4, which clarifies the relationships between the root causes of inadvertent DADDs and their triggers. These relationships have been applied to all inadvertent DADDs, which are mentioned by the interviewees. The temporal relationships between concepts (e.g. process steps in Fig. 3) have been decided based on the provided timeline of each architectural design scenario, as mentioned by the interviewees. The timeline of each scenario has been captured using temporal words and phrases such as "initial", "then", "first", "a month behind", "later", "in the short term". For example, one interviewee said "*first come up with the proper solution*". This indicated that the first step is to identify a maintainable solution, rather than a debt-incurring solution.

We performed the three grounded theory analysis steps iteratively across the transcripts of the interviews, the focus group, and answers of follow-up questions; the analysis of each transcript (pertaining to an interview or the focus group or follow-up questions) presents an iteration. Within each iteration, the first author (an experienced researcher) applied the three grounded theory analysis steps. The industrial and research experience of the first author supported better conceptualization and establishment of relationships between concepts (i.e. *theoretical sensitivity*), as well as developing cohesive categories (i.e. *cohesive theory*).

By the end of each iteration, we identified conceptual gaps, which are not sufficiently covered from the gathered data, and consequently collected more data until reaching *theoretical saturation*. When collecting new data, we applied *theoretical sampling*. In other words, we decided to further interview practitioners (either from the embedded or enterprise domain, as explained in Section 2.3) based on the conceptual gaps in results. For instance, interviewees from the embedded domain provided more concepts regarding deliberate DADDs (see Section 4), but less concepts regarding inadvertent DADDs (see Section 5). This has been compensated by the interviewees from the enterprise domain, who provided more concepts regarding inadvertent DADDs.

The result of this phase is an initial list of concepts and their relationships. We used the Atlas.ti[3] qualitative analysis tool to facilitate coding. Moreover, we provide our initial and final codes in the replication package[2].

2.4.2. Ensuring agreement on annotations

In research methods involving qualitative content analysis, reliability of the analysis is critical. To strengthen reliability, the first and third authors conducted two iterations of reliability tests: The third author independently annotated selected sentences (that were also coded by the first author) and compared it to the original annotations of the first author; disagreement was discussed and the groups of annotations were modified until consensus was reached.

3. RQ1 - Types of DADDs

In this section, we classify *all the identified* DADDs into specific types, based on a well recognized taxonomy of design decisions [11], and taking into account the reasons for incurring technical debt. The types of DADDs have been derived by analysing the design scenarios mentioned by the practitioners during the interviews as explained in Section 2. In the following sub-sections, we explain the types of DADDs, and support them with examples. The examples are structured according to the following annotated parts: [Design issue], [Solution alternatives], [ADD rationale] and [Extra effort].

3.1. Existence DADDs

Existence decisions are concerned with adding new elements (e.g. components or dependencies) to an architecture. Existence decisions could be structural or behavioural. Structural decisions add components or dependencies to one of the views of the architecture [11]; for example, a system should have an API component to accept customer requests. Behavioural decisions add new interactions between components to implement the functional and non-functional requirements [11]. Existence decisions are well known to practitioners, and are considered to be the most commonly made design decisions in practice [23]. According to the scenarios provided from practitioners, we found that existence DADDs incur either requirements or architectural debt. Both could be incurred deliberately (see Section 4) or inadvertently (see Section 5). We explain both cases in the following sub-sections.

3.1.1. Existence ADDs which incur requirements debt

These are existence design decisions, which only partially achieve current or upcoming architecturally significant requirements (ASRs) of a software system. The non-fulfilled requirements could be functional as well as non-functional requirements. As a result, system evolution (implementing new features) becomes cumbersome, and requires significant efforts to refactor or rewrite existing system architecture to fulfil missing requirements. In the following paragraph, we provide an example from one of the practitioners:

Example. An existence ADD, which partially fulfils security and usability requirements

[2] https://github.com/m-a-m-s/DADDs.

[3] https://atlasti.com/.

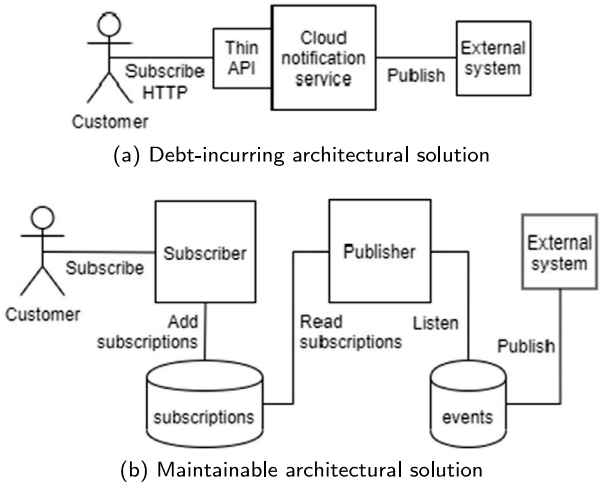


Fig. 2. Example of a maintainable and a debt-incurring solution.

A system requires to implement a publish–subscribe mechanism, where customers can subscribe to certain events, and receive notifications [design issue]. Two architectural solutions have been proposed (see Fig. 2): a maintainable and a debt-incurring solution [alternative solutions]. Both solutions satisfy the functional requirements. However, the debt-incurring solution depends on using a cloud notification service, which imposes limitations regarding security and usability. For example, the cloud notification service supports only anonymous authentication and provides no ability to store specific authorization rules for each user. Moreover, users must follow complex steps to deal directly with the cloud notification service. On the other hand, the maintainable architectural solution provides a customized solution, which contains an authentication mechanism and a dedicated user interface with better usability. Nevertheless, the debt-incurring solution has been selected, because it could be implemented in a shorter time period, even though it does not satisfy the security and usability requirements [ADD rationale]. As a consequence of this decision, it would not be possible to fully satisfy security or usability requirements (e.g. re-playing events per user, which depends on the user security profile) without re-engineering the debt-incurring solution to the maintainable solution [Extra effort].

3.1.2. Existence ADDs which incur architectural debt

These are existence ADDs, which do not follow certain architectural principles; this consequently deteriorates the software architecture. The practitioners mentioned scenarios about breaking four architectural principles: separation of concerns, conceptual integrity, components dependencies, and components abstraction (for a description of these principles, see [12]). This type of DADD has a direct influence on the maintainability of the system, such that the costs of adding extra features to the system, and the costs of refactoring DADDs would gradually increases by time [8,24].

Example. An existence ADD, which break the separation of concerns between components

A software system is divided into multiple components, where separate teams are responsible for each component. For some features, you need to change multiple components, and different teams must adapt to new interfaces [Design issue]. Two architectural solutions were proposed: A debt-incurring solution to implement all functionalities in a single component (rather than in their relevant components), and a maintainable solution to implement each functionality inside its relevant architectural component (potentially by different teams) [Solution alternatives]. The debt-incurring solution has been selected, because one of the teams was busy implementing other functionalities, and this would have delayed the delivery of the software [ADD rationale]. As a consequence, the understandability and maintainability of components will degrade, and will require re-engineering these functionalities to move them later to their responsible components. [Extra effort]

3.2. Technology DADDs

Technology ADDs are choices regarding the adoption of certain technologies, tools and products [25]. Technology decisions can be either made by software engineers or enforced from the business environment (the latter is termed *executive technology ADDs* in [11]). Selecting the wrong technology solution could lead to incurring technical debt. Based on analysing the scenarios from the practitioners, we determined two types of technology DADDs: executive technology DADDs, and build-versus-reuse technology DADDs. We explain both types in the following sub-sections.

3.2.1. Executive technology DADDs

Selecting an optimal technology solution requires comparing technologies regarding their benefits and drawbacks [25]. However, due to business reasons, a company or client might force software engineers to select certain technology solutions without comparing it with other solution alternatives. The enforced technology solution might not be the most optimal solution, and might involve significant drawbacks, which could lead to the following consequences:

- Due to technology drawbacks, new software features cannot be implemented without replacing the selected technology with another. This is a significant change in a system, and requires extra effort to implement new features.
- To overcome the drawbacks of a technology, sub-optimal workarounds must be implemented. This has a direct impact on the maintainability of a software system, because such workarounds are usually bug-prone.

According to the scenarios provided from practitioners, we found that executive technology DADDs could be incurred both deliberately (see Section 4) or inadvertently (see Section 5). If executive technology DADDs are made deliberately, then the technical team is aware of the drawbacks of this technology. However, they could not convince business managers to select another technology. To clarify executive DADDs, we provide an example from our interviews for an executive DADD that requires implementing workarounds to overcome its limitations.

Example. An executive technology ADD, which requires implementing workarounds to overcome its limitations

A system requires storing files in the cloud. The system needs to retrieve the files efficiently for customers based on their profile [design issue]. There were two options, (1) use an in-house content management technology, which lacks features to store meta-data, and lacks an efficient searching mechanism, or (2) use third party content delivery network technologies to store files in a distributed and efficient way [solution alternatives]. It has been decided to use the in-house content management technology, because this aligns with the company policy to use in-house software, and minimize using 3rd party software [ADD rationale]. As a consequence, using the content management system required developing workarounds to overcome the lack of features. This became a source of bugs in future releases of the system [extra effort].

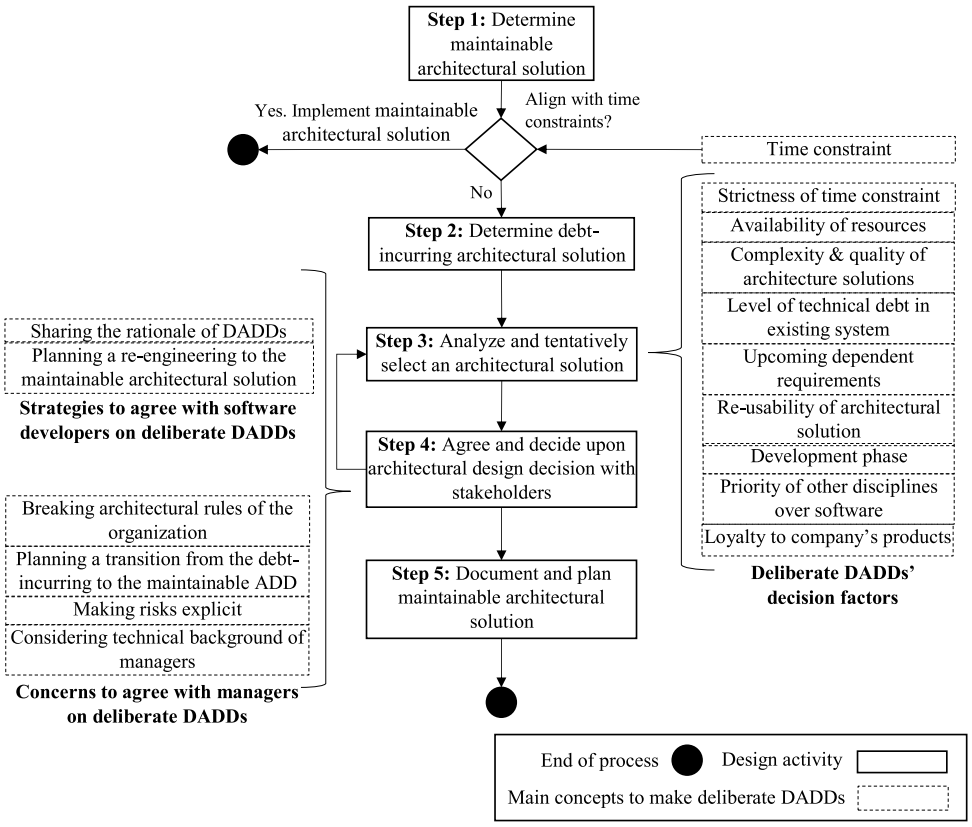


Fig. 3. Deliberate DADDs decision making process.

3.2.2. Build-versus-reuse technology DADDs

Re-using an existing technology solution (e.g. a library or a framework) saves a considerable amount of time and effort to develop a software system. Therefore, it is common that software engineers decide on re-using existing technologies, which are created by other teams within the same company or by external technology vendors. However, before deciding to re-use an existing technology, software engineers might consider building the functionality themselves instead of re-using (or buying [26]) it. This might be due to high costs of technology solutions, or due to existing technical drawbacks of technologies (e.g. lack of features or performance problems), or due to delays in the release for certain versions of a technology.

Deciding between building or re-using a technology is a critical decision, which might incur technical debt. On the one hand, software engineers might prefer to build a technology themselves (instead of re-using an existing one), which causes significant extra effort to re-develop and maintain a separate version of an existing technology with a good quality. On the other hand, selecting a technology solution with a drawback can negatively affect the evolvability of a system (as explained in Section 3.2.1). In other words, it can incur significant costs to replace or even re-build a technology due to its drawbacks.

According to the scenarios provided from practitioners, we found that build-versus-reuse technology DADDs happen mostly inadvertently (see Section 5). This might be due to the complexity of this type of decisions (i.e. build-versus-reuse), which make it harder to foresee their consequences. To clarify build-versus-reuse technology DADDs, we provide an example from one of the practitioners in the following paragraph.

Example. Build a technology instead of re-using it from another team in the same company

A business application should communicate with an external service using a certain protocol [design issue]. A library is required to perform the communication with the service. Two main options were proposed. The first is to use a library provided by another team in the same company; however, this library was still missing important features to comply with company regulations. The second was to develop an own version of the communication library [solution alternatives]. The team decided to develop their own communication library, because it is faster to deliver the software [ADD rationale]. As a consequence, the team maintained this communication library duplicating the maintenance effort, as both them and the other team had to resolve similar problems. Moreover, it would require re-engineering to switch later to the original communication library from the other team [extra effort].

4. RQ2 - Making deliberate DADDs

Based on the gathered data from the interviewees, we derived a decision making process comprised of five main steps to make deliberate DADDs. Fig. 3 shows these steps and their dependencies. We note that the application of these five steps is subjective, as it depends on the knowledge, experience, expertise and skills of software engineers; of course this is the case for any kind of architecture decision making [27]. We explain each of these steps in the following sub-sections.

4.1. Step 1: Determine maintainable architectural solution

In this step, software engineers propose an architectural solution for a design issue, which satisfies the maintainability requirements of a software system (i.e. it does not incur technical debt). Of course, the solution also fulfils other relevant functional and non-functional requirements. Moreover, determining a maintainable architectural solution might require making trade-offs between the different non-functional requirements. An example of a design issue mentioned by interviewee 1: "*we develop software in multiple components and teams. In case software solution has an impact on multiple teams, then things become*

more complex to implement...this will cost more effort than when they are just in one simple team...because you need to change the interfaces and both teams can adapt to these new interfaces.". For this design issue, a well maintainable architectural solution is to follow the separation of concerns and implement each functionality inside the relevant architectural component (potentially by a different team). This solution requires higher effort as the different teams need to synchronize when implementing their own part, but it does not incur debt.

In this step, software engineers strive to propose an architectural solution that does not deliberately incur technical debt. However, they might select an architectural solution that incurs inadvertent technical debt. This has several root causes such as lack of architectural skills or lack of alternatives. In Section 5, we elaborate more on inadvertent DADDs, their root causes and triggers.

4.2. Step 2: Determine debt-incurring architectural solution

If the effort required to implement the maintainable architectural solution (from step 1) does not align with the time constrains of the project (e.g. time to market), then the team proposes a debt-incurring architectural solution, which could be implemented in a shorter time period. However, a debt-incurring architectural solution has a negative impact on the maintainability of a software system on the long term. An example of a debt-incurring architectural solution for the design issue mentioned in step 1 is to implement all functionalities in a single component (rather than in their relevant components). This will allow faster delivery of functionality (as only one team is involved) but will deteriorate maintainability (as one component becomes overly complex and has many dependencies with other components).

When working on this debt-incurring architectural solution, software engineers strive to achieve two main goals:

1. *Fulfil customer visible requirements*: It is expected that the debt-incurring solution fulfils all visible customer requirements (i.e. functional requirements, and run-time quality attributes).
2. *Localize incurred technical debt*: The architectural solution that incurs technical debt should be encapsulated in a limited number of components. This would facilitate the re-engineering to a maintainable solution in a future release.

4.3. Step 3: Analyse and tentatively select an architectural solution

In the previous two steps, two architectural solutions have been proposed: a maintainable and a debt-incurring one. In this step, software engineers select tentatively one of these two solutions. This selection is influenced by several factors:

- *Strictness of time constraints*: Software engineers tend to prefer the debt-incurring architectural solution (from step 2) over the maintainable architectural solution (from step 1), if the time constraints are firm and critical to the company. For example, a fixed date for market introduction before competitors can be a strict deadline.
- *Availability of resources*: If some software developers are busy with other tasks, it may not be feasible to select the maintainable solution. For example, in the design issue mentioned in step 1, if one of the teams is busy developing other functionality, and cannot participate in developing the proposed maintainable architectural solution (from step 1), then the debt-incurring solution would be preferred (from step 2).
- *Complexity and quality aspects of the proposed architectural solutions*: The difference in complexity and quality (and consequently the effort) between the maintainable and the debt-incurring architectural solutions play a big role. For example, a localized debt-incurring architectural solution is more acceptable, because it could be easily changed in further iterations. On the other hand, a maintainable solution that is too complex to implement may not be easily agreed upon.
- *Level of technical debt in existing system*: The diffusion of technical debt items (e.g. code smells) in the whole system could be a factor to prefer the maintainable architectural solution (from step 1) over a debt-incurring architectural solution (from step 2), esp. if maintainability is already widely compromised. For example, one interviewee explains the factor to decide on a maintainable architectural solution "*we saw much legacy code and redundant obsolete code...we took the decision to build a new architecture*".
- *Upcoming dependent requirements*: In case new requirements in the next iterations are meant to build upon the currently decided architectural solution, then the maintainable architectural solution would be preferred over the debt-incurring architectural solution; the reason is that the maintainable solution can facilitate the implementation of those requirements.
- *Re-usability of architectural solution*: The intention to re-use the developed software components in other products would require developing a stable and maintainable architecture for the product, rather than a fragile and hard to maintain architecture.
- *Development phase*: In an early iteration of a project, software development teams strive to achieve customer satisfaction by efficiently delivering implemented requirements; thus they prefer a debt-incurring solution over a maintainable one. On the other hand, in later iterations of the project, when the size and complexity of the software have expanded, extra care is required for the maintainability of the software.
- *Priority of other disciplines over software*: In the development of embedded software, the quality of software could play a secondary role in the development of software compared to other engineering disciplines. For example, innovations in chemical engineering has a higher priority over software in some embedded systems. As a result, software development tries to support other disciplines in their experiments using quick solutions rather than developing well-maintained and long-term solutions.
- *Loyalty to company's products*: Software engineers could be biased or forced to choose their company's own products, even though the selected products might not be the optimal solutions for the issue at hand. One interviewee explained the reasons for this: "*It was because of the company direction, and policy, and to work as one team...we do not want to introduce security issues*". Consequently, software engineers try to adapt and customize these products to overcome their limitations. This causes several workarounds and shortcuts.

4.4. Step 4: Agree and decide upon architectural design decision with stakeholders

ADDs are taken in a group, and require agreement among different stakeholders [28,29]. In this step, decision makers try to agree on the tentatively selected architectural solution (from step 3) with pertinent stakeholders. The interviewees mentioned two main stakeholders to discuss with and reach agreement: software developers and managers. In the following sub-sections, we explain the agreement with software developers and managers on DADDs.

4.4.1. Agreement with software developers on DADDs

Ensuring the agreement with software developers on an ADD is important for the seamless implementation of ADDs [29]. Debt-incurring ADDs are challenging to agree upon with software developers, as developers prefer to maintain the good quality of their system. Decision makers use two strategies to achieve an agreement with software developers on a debt-incurring ADD:

- *Sharing the rationale of debt-incurring ADD*: By understanding the rationale behind the decision, software developers can realize the problem from the perspective of decision makers. Moreover, sharing the rationale of ADDs gives software developers a motivation

to implement the taken ADD and achieves a successful delivery for the product. One interviewee mentioned "*It is important to share the rationale behind the decision...it makes it easier to accept and share the approach later to get rid of it. If you share that it makes easier to accept...if you are open of the rationale and why of the decision. You have a big step to get everybody on board*".

- *Planning a re-engineering to the maintainable architectural solution*: promising software developers to pay back the technical debt in a following iteration supports the agreement further. Decision makers make a plan with software developers to first implement the debt-incurring architectural solution and change it at a given point of time to the maintainable architectural solution. One interviewee mentioned "*We must explain to designers, we are not taking the right decision but you must understand that we have to deliver and comply to planning...they accept that based on the intention that we make the proper solution later*".

4.4.2. Agreement with managers on DADDs

Aligning with the plan of the project and policies of the organization is the main concern of business and project managers. Agreement with managers is crucial when deciding on an important architectural decision, which has a significant impact on the product and customers. However, it is quite challenging for software engineers to communicate with managers and persuade them for an ADD, which could impact either the plan of the project or the standards of the organization. An interviewee spoke about convincing managers to agree on a maintainable solution "*we need to run in the politics of the organisation to explain that we need to invest much development time*". Another interviewee said "*You need to find your way in the organisation*".

Software engineers consider the following concerns when agreeing with managers:

- *Breaking architectural rules of the organization*: when deciding on a debt-incurring ADD, it might be required to break one of the main architectural rules in the organization. For example, one interviewee mentioned "*on the department level, they want to have good quality and the design and reference architecture...if you make big shortcuts on that, you need to have deal on the department managers. If you overtake the reference architecture regarding hardware, these are big changes, then you need to convince your department*".
- *Planning a transition from the debt-incurring to the maintainable ADD*: To mitigate the risk of paying interest for technical debt in future development iterations, software engineers agree with managers on a future plan to re-engineer the debt-incurring solution (from step 2) into the maintainable solution (from step 1). This requires agreement on further costs and resources in the next development iterations. Practitioners discussed this problem during the focus group:
  "***Interviewee 1**: we make deal with the manager, we make this shortcut for the project now but we need more people and more time to do it in a better way later.*
  ***Interviewee 2**: We had the same discussion, we took technical debt to create product...and in next iteration plan, we will resolve this technical debt*"
- *Making risks explicit*: Managers might not be aware about the consequences of DADDs on the quality of a software. On the one hand, the maintainability of the product will degrade, when making a debt-incurring ADD. This might impact the speed of development in following iterations due to paying technical debt interest. On the other hand, a debt-incurring ADD might be a cause for bugs in future releases, which would impact the external quality of the product. It is the responsibility of software engineers to make the risks of DADDs, as well as their consequences as explicit as possible to managers.
- *Considering technical background of managers*: The technical background of managers plays an important role to persuade them for accepting or preventing DADDs. One interviewee mentioned "*Everyone can see short term cost. Long term costs are really difficult for them to see. You cannot cost (i.e. calculate) technical debt accurately...This is a big problem with technical debt, and the real hard architectural debt is very hard to cost. Thus, the people who controls the budget should have a good technical understanding... They will know then the risk they will take*".

4.5. Step 5: Document and plan maintainable architectural solution

When deciding on a debt-incurring ADD, software engineers also document the maintainable architectural solution (from step 1) to be considered in a future debt remediation. For example, interviewee 1 mentioned "*when we do a short term solution, I always add additional re-engineering work items in the backlog to reduce it...I tag them with technical debt*". The usage of issue tracking systems is convenient for software engineers to track and plan tasks for paying back technical debt items. However, software engineers also mentioned that they do not document the rationale behind taking a debt-incurring ADD. The realization of a remediation plan depends on several factors, which are discussed in Section 6.

5. RQ3: Understanding the manifestation of inadvertent DADDs

An inadvertent DADD is a design decision, which had no foreseen negative consequences on system maintainability at the time it was made. However, technical debt manifested later due to the occurrence, first of certain *root causes*, and subsequently of unforeseen *triggers*, which release the technical debt (the DADD becomes activated). Fig. 4 shows the root causes and triggers of inadvertent DADDs that we found in our study, and illustrates *how* inadvertent DADDs occur. We note that all inadvertent DADDs scenarios provided from the practitioners show to have a single trigger but one or more root causes. However, we have generalized the concept in Fig. 4 to consider the case of having multiple triggers. Moreover, we note that triggers could activate DADDs that were previously unknown. Finally, we note that multiple co-occurring root causes could contingently lead to a single inadvertent DADD.

Both the root causes and triggers must occur in order to incur technical debt. In some cases, triggers release technical debt *directly or shortly after* the occurrence of root causes. For example, when a technology solution with drawbacks is selected due to lack of architectural skills; the technology drawbacks can directly cause extra costs during the development phase. In other cases, triggers release technical debt *long after* the occurrence of root causes. For example, when a technology solution is selected carefully, but the technology provider cancels support for this technology a few years later. In this case, technical debt is released once the lack of support hinders making changes to the system.

In the following sub-sections, we explain the main concepts (root causes and triggers) of inadvertent DADDs, and provide three examples from the practitioners that illustrate how root causes and triggers are combined.

5.1. Root causes of inadvertent DADDs

Root causes of inadvertent DADDs do not directly cause technical debt items. However, they are reasons for making certain (sub-optimal) ADDs, which later (once certain triggers occur) become debt-incurring. To answer *why* inadvertent DADDs occur, we have identified eight root causes of inadvertent DADDs, as explained in the following sub-sections.

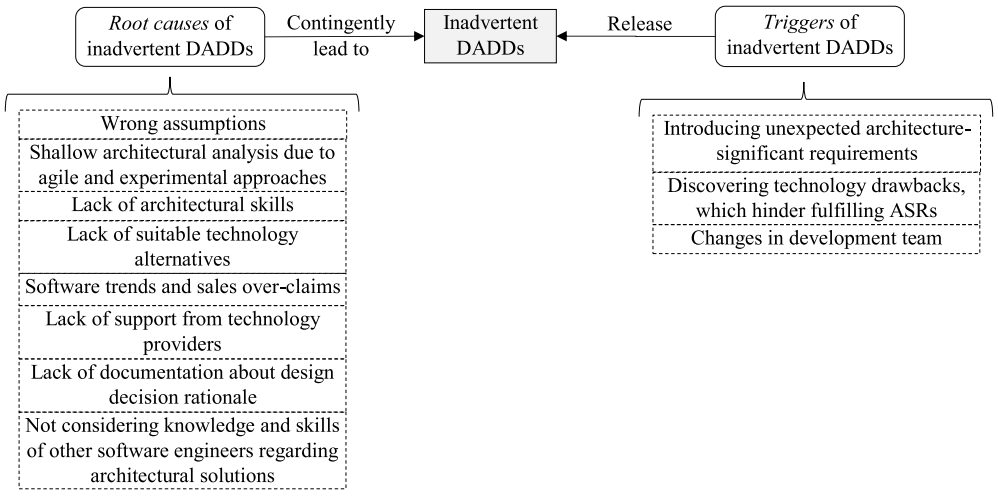


Fig. 4. Inadvertent DADDs manifestation model.

5.1.1. Root cause - Wrong assumptions

When deciding on an architectural solution, software engineers might make assumptions, i.e. they deal with unknowns through assuming facts without proof [30]. Developing software in an interdisciplinary environment increases the risk of making the wrong assumptions, thus missing important ASRs. This is due to the difference in knowledge and way of communication (e.g. using terms with different meanings) between different disciplines, which makes software engineers misunderstand or mis-communicate information from other disciplines. For example, interviewees discussed challenges of software engineers in the embedded domain:

"***Interviewee 1****: you think you know how the machine works and which parts are connected to each other, based on that you make the decomposition. Later on we discover an electrical interaction which we miss…in multidisciplinary environment, people have different language.*

Interviewee 2*: It is more about the assumption, about the expected behaviour for the system you are developing. You have certain assumptions, this is how it should work. And these assumptions are not always right, and based on the assumptions, you make certain decisions.*

Interviewee 3*: Sometimes lack of communication with different domains… Not knowing exactly from each other which information to share*".

5.1.2. Root cause - Shallow architectural analysis due to agile and experimental approaches

Market analysis and product development might follow an experimental and iterative approach. First, requirements are gathered from customers in a short time period (e.g. few months for market analysis). Second, a product is developed based on the gathered requirements. Third, customers use the product and provide feedback on their experience with the product and possible new requirements. Finally, customer experiences are considered and the product is modified.

While an agile approach is efficient for product development and effective for selling products, it increases the risk of missing ASRs, and consequently introduces debt-incurring architectural solutions in following iterations. One interviewee gave an example about the impact of the product development process on the quality of the product "*You learn something from customers. The machine is in a completely new market we do not know our customers. We do not know how they use that machine. Sometimes we learn from them. They do that I never thought about it… As a result, they got all kinds of issues which we did not foresee upfront…we need to change many things which are not in the design…we are still solving many issues because of this change*".

5.1.3. Root cause - Lack of architectural skills

The skills of a software architect are many and diverse and include both technical, as well as soft skills [1]. The lack of architectural skills could be a reason for making sub-optimal ADDs, and consequently incurring technical debt inadvertently, when inadvertent DADD triggers occur. One interviewee mentioned "*One huge problem, in term of the architectural process, you always have options, and within your options, you have pros and cons for every single option, and you got to get people at that stage. That requires skills, experience and discipline. Not many architects have all three…but going through all the pros and cons is difficult. There are few people who can do that at a high standard*"

5.1.4. Root cause - Lack of suitable technology alternatives

To decide on a technology solution (e.g. library or framework), software engineers compare technologies with each other to determine their benefits and drawbacks [25]. Due to the lack of suitable technology alternatives, software engineers are sometimes forced to decide on a solution with potential drawbacks, which can threaten the maintainability of a system in future changes, and incur technical debt. One interviewee mentioned "*I thought that this was the best solution because I had compared other communications mechanisms…but you had to select a solution, because this is the one available*".

5.1.5. Root cause - Software trends and sales over-claims

Software engineers might decide on a certain architectural solution, because it is the current trend at that time, or because it has been advertised for being a prime solution. However, once the solution is implemented and the software starts to evolve, the drawbacks of the solution surface, and technical debt is incurred. An interviewee provided an example and said "*A technology vendor invented their own business rules engine…the calculation engine was happening in a proprietary language. This was terrible and nightmare language, and it never worked. It was a great sell pitch…they could never change business rules as easy as they claim. This is massive inadvertent technical debt*".

5.1.6. Root cause - Lack of support from technology providers

When technology providers decide to stop evolving and supporting certain technology solutions, technology solutions which are used in many existing software systems become obsolete. This is a well-known root cause for incurring inadvertent technical debt [6].

At a given point in time, software engineers discover an unsolvable drawback (e.g. a security threat) in an obsolete technology. It is then required to cope with technological changes by replacing this obsolete technology with an up-to-date one. One interviewee recommended monitoring the evolution of technologies "*Certain 3rd party libraries can become out of date and need to be refreshed, due to open source security stuff, you need to monitor and we need to track that somehow*".

5.1.7. Root cause - Lack of documentation about design decision rationale

When changes happen in the software development team, new team members are not aware of the rationale behind existing ADDs. One interviewee spoke about the loss of architectural knowledge of ADDs: "*if you switched the team, they will miss a lot of information. There is lot of rationale, which we do not document*". By losing the rationale behind existing ADDs, it becomes harder for new developers to understand existing ADDs within the system (e.g. why they were made and what were the alternative solutions). This renders new developers unprepared and uninformed for making changes to existing ADDs. This subsequently results in making changes, which unintentionally deviate from the intended purpose of the architecture, and make future changes harder to implement.

We note that this problem also appears with deliberate DADDs (see Section 4), which have been implemented, but they have not yet been removed from the system. In this case, new developers will not be aware about the plan to re-engineer existing deliberate DADDs. This often results in building further upon existing DADDs, and incurring extra technical debt in future changes.

5.1.8. Root cause - Not considering knowledge and skills of other software engineers regarding architectural solutions

Software engineers might tend to decide on architectural solutions, which solely fit their own knowledge, skills and perspective. However, when the team changes, it becomes challenging for new team members to understand and maintain the existing design of a system, because the new team members are not familiar with the implemented architectural solutions. One interviewee mentioned an example from his experience "*if you're familiar with the Scala programming language…it has so many features that interact in such strange ways that you can look at a piece of code and have absolutely no idea how that works and I think that there are certain developers who…will use the features to the extent they can…I was working on building a Pipeline and the structure of the pipeline was such that everything could be reduced into three Lambdas…when I built it it was completely understandable…but I have heard from the people…that it over-abstracted the problem…now everybody who has to touch that piece of code…you have to understand the indirect Lambda layers…I think I'm coming to the point where that causes a lot of technical debt those sorts of decisions, not recognizing that the code has to be handed off to other people*".

5.2. Triggers of inadvertent DADDs

Triggers of inadvertent DADDs are unforeseen events that lead to incurring technical debt, due to certain root causes. We have identified three types of triggers, which create technical debt items, as elaborated in the following sub-sections.

5.2.1. Trigger - Introducing unexpected architecture-significant requirements

Software engineers try to design a sustainable architecture, which supports effective and efficient changes to the system without degrading the internal quality of the system, i.e. without incurring technical debt. For example, one interviewee said "*From the development team, it is never a choice to take a technical debt…we don't want this technical debt. we want to have the correct solution*".

This requires to determine upfront, all potential future architectural significant requirements (ASRs), in order to design a flexible system, which can adapt to changes. However, this is not feasible and eventually unexpected ASRs will occur. In these cases, software engineers may need to perform significant changes to the system architecture to accommodate such ASRs. One interviewee spoke about building a new architecture because of new requirements "*we made a new design for this feature…this was a new requirement, which we did not know when we started the project…you can not make a complete design which suitable for many years because you will always get new requirements which we did not take into account*".

Unexpected ASRs are not only functional requirements (e.g. adding a new feature to the product) but also quality attributes, which did not have sufficient priority during the initial design of the software architecture. For example, one interviewee explained a scenario about an unexpected re-usability requirement: "*a component was designed to use local services…we have colleagues, who also make products in another location, and they can use the same component… At the beginning it was easy without their re-use, now we have…technical debt, to share a component with somebody else*".

5.2.2. Trigger - Discovering technology drawbacks, which hinder fulfilling ASRs

The decision on software technologies depends on evaluating technologies according to their potential to fulfil architecture significant requirements (ASRs). To do this, software engineers compare alternative technologies according to their benefits and drawbacks, and then choose a technology solution that optimizes benefits vs. drawbacks, regarding the fulfilment of ASRs [25]. However, it is challenging for software engineers to determine all benefits and drawbacks of a technology solution upfront (i.e. during making the design decision). Thus, software engineers might discover technology drawbacks after selecting and implementing a technology solution. If the discovered technology drawbacks prevent fulfilling ASRs, then software engineers might need to implement complex workarounds to overcome these drawbacks, or in the worst case replace the whole technology with another. These are extra maintenance efforts to maintain and evolve the system.

5.2.3. Trigger - Changes in development team

A common phenomenon in software development is the continuous turnover of software developers. For example, software developers who participate in the initial development of a software project often leave the team during its maintenance phase. Losing experienced team members from a project entails also losing their knowledge about architectural design decisions within the system, and specially the rationale behind making these decisions, which are rarely documented [16].

5.3. Examples of inadvertent DADDs

In this section, we present three examples of inadvertent DADDs, as provided by the interviewees. For each example, we pinpoint the root causes and triggers of the inadvertent DADDs. Moreover, we support each example with quotes from the practitioners to further illustrate the manifestation model of inadvertent DADDs as presented in Fig. 4. The quotes are structured by using the annotations [Root cause], [Trigger], and [Inadvertent DADDs], right after the corresponding parts.

- **1st Example**: In this example, the root cause "*Shallow architectural analysis due to agile and experimental approaches*" is combined with the trigger "*unexpected architecture significant requirements (ASRs)*" to create inadvertent DADDs.
  "*The reason is that you make a design at the moment when you know a number of requirement but probably you miss one…We also we don't know how they use that machine [Root cause]…we did not sell a double machine, but customer see that it is possible technically. This is a nice feature [Trigger]……but if we have to redo that we need to change number of things which are not in the design, and we did not put them in the design because we didn't sell the machine as double machine [Inadvertent DADDs]…we will learn from customer much quicker and know what to improve. So try to make those cycles very short. Agile way of working [Root cause]*"

- **2nd Example**: In this example, the root cause "*lack of suitable technology alternatives*" forces software engineers to select a technology solution with potential drawbacks. Once the "*technology drawbacks are discovered*" (trigger), the inadvertent DADDs are released and manifested.
  "*what they did was to make a fork from a GitHub project, and made the changes and use the fork [Inadvertent DADDs], but they never push the changes back or kept up to sync with the original open-source library…it's actually a library that is implements a certain standard, so if something is added to a standard and the original author puts it in his library, we don't automatically have it, so we need to invest time to add the standard to our modified library [Trigger]…Q: At that point of time there was no other alternative, this was the only alternative which existed? A: As far as I know in the stack and the language we are using, yes [Root cause]*"
- **3rd Example**: In this example, the root cause "*Not considering knowledge and skills of other software engineers regarding architectural solutions*" is combined with the trigger "*Changes in development team*".
  "*when I built it it was completely understandable, and cut out hundreds of lines of boilerplate, this was in Java [Inadvertent DADDs]…I think that's an example of trying to be clever trying to boil a problem down to a very abstract form that adds technical debt [Root cause], because now everybody who has to touch that piece of code… it's the core of the application, if you extend it adding another transformation, you have to understand how that table is structured [Trigger]…I have had to apologize for that code*".

6. RQ4 - Dealing with DADDs

According to the study subjects, DADDs have two main implications:

1. *Increase in effort of making changes to the system*. For example, it is harder to add new functionalities, as Interviewee 4 pointed out: "*the more technical debt you add to the system, the slower the next feature addition will be. In the end…We need to do something or our progress will completely stand still*".
2. *Appearance of bugs in production*. Interviewee 1 explained the impact of DADDs on user functionality "*we get…bug reports…which has a relation to this technical debt…then we remember. We did make a quick fix on that*". Bugs appear due to the different types of DADDs (see Section 3). For example, Interviewee 1 spoke about missing scenarios when analysing user requirements: "*You have a quick fix… But…issues will be introduced in that way. So you miss interactions*". In this scenario, the increase of bugs happens when implementing existence ADDs, which incur requirements debt.

The subjects distinguish between the implications of deliberate vs. inadvertent DADDs. On the one hand, deliberate DADDs can be well planned, localized (as explained in Section 4), and consequently better tested. This mitigates the risk of having significant bugs in production. One interviewee mentioned "*This kinds of issues I haven't seen them come from technical debt which we deliberately introduced…I think that short term solutions are never that bad that we have critical bugs in the field*". On the other hand, inadvertent DADDs cannot be planned or expected. Thus, their implications are usually more serious than deliberate DADDs. For example, one interviewee speaks about the impact of inadvertent DADDs due to an unexpected ASR: "*sometimes customers face problems which we did not see…that we cannot solve quickly*".

Dealing with DADDs happens either pro-actively or re-actively. The proactive approach applies to deliberate DADDs, by planning upfront to re-engineer the implementation of DADDs to a maintainable solution as explained in Section 4. On the other hand, the reactive approach is initiated when software development faces maintainability issues or bugs, which significantly impact the quality of software. Interviewee 1 discussed the reactive approach: "*At a certain moment in time…technical debt slowly grows…you need to take measures to get back to a descent level*".

When dealing with DADDs, the development team needs to decide between either maintaining or re-engineering the implementation of the DADDs. The decision on which of these two approaches to follow is taken in a group (similarly with ADDs). Interviewee 2 mentioned how they decided to maintain DADDs in a software project "*There was an educated guess, we did it with knowledgeable developers in that area. To see the pros and cons in development time and impact of regression*". Several factors influence the decision between maintaining or re-engineering the implementation of DADDs:

- *Availability of skilled resources*: The availability of time and resources are important decision factors (as explained in Section 4). In addition, re-engineering is a challenging task, which requires knowledgeable and skilled resources. Interviewee 1 mentioned "*The teams have been decreased…no specialist knowledge anymore within the teams and people are eager to choose a quick fix more often…it is about not having the right knowledge to take the good decisions anymore*".
- *Project phase*: The possibility to conduct a re-engineering activity differs between project phases:
  - At an early phase, the team is keen on developing new features rather than re-engineering existing ones.
  - After further iterations of software development, software engineers are more keen to conduct re-engineering activities in order to facilitate future iterations.
  - During maintenance phase, lack of sufficient and knowledgeable resources prevents making significant changes to existing functionality; so re-engineering is again avoided.

  Interviewee 1 mentioned "*it really depends on the timing of the project. In a first stage of the project. First market introduction, then pressure is high…after that you have a stable platform… From a software point of view. We have time to fix technical debt or update the design of the current status of the engine and the expected changes in the near future*"
- *Age of DADDs*: Recently taken DADDs have higher chances of being re-engineered than those taken earlier, as the latter might be harder or less important to change. Thus, software engineers try to re-engineer DADDs as fast as possible, otherwise it becomes harder to change them later. For example, interviewee 3 mentioned "*That's the risk of postponing proper design in the future. In future, team could change*".
- *Importance of the project*: An important software product with future plans deserves investment in its quality than a product with potentially minimum future changes.
- *Risk of existing technical debt*: The impact of DADDs on maintainability and indirectly the occurrence of bugs plays an important role on investing to conduct a re-engineering activity. Interviewee 2 said "*It depends on the impact of the technical debt but I always try to get the most risky stuff as soon as possible*".
- *Company policy*: The company policy provides the motivation and support to deal with technical debt. On one hand, some companies have a specific process and budget for resolving technical debt. On the other hand, other companies (e.g. consulting companies) might profit from the availability of technical debt, because customers might pay extra costs for maintaining their systems.
- *Complexity to identify and agree upon technical debt*: It is very challenging for software engineers to identify problems in the architecture of an existing system, because architectural debt might not be directly visible and might reside on different abstraction levels (e.g the conceptual level of an architecture). Moreover, it could be challenging for software engineers to agree upon the problem causing the technical debt.

7. Discussion

In this section, we discuss our results regarding their implication, both for researchers and practitioners. We argue that our results could support researchers and practitioners to achieve two goals: capturing architectural knowledge and making DADDs. We discuss these two goals in the following sub-sections.

7.1. Capturing architectural knowledge

7.1.1. Implications for researchers

On the one hand, capturing relevant architectural knowledge (AK) from an existing system through documenting its ADDs [31], supports maintaining and involving this system by modifying the ADDs or making new ones. On the other hand, capturing AK supports the re-use of AK (e.g. from developer communities [32] or issue tracking systems [33]) to design new systems.

To capture AK, researchers proposed taxonomies of ADDs (e.g. [11]) and AK models (e.g. [34]), which provide the foundation for AK capturing approaches. Our results extend this foundation with additional concepts regarding DADDs. Such concepts include the types of DADDs (see Section 3), decision factors for making deliberate DADDs, and concerns considered when working on agreements for deliberate DADDs with stakeholders (see Section 4). Specifically, our contributions support extending current AK capturing approaches in the following ways:

- *Documenting DADDs*: Current ADDs documentation approaches (e.g. [31]) could be extended to document DADDs. This extension should distinguish the types of DADDs from each other (see Section 3). Moreover, it should allow documenting both debt-incurring and maintainable solutions, as well as re-engineering plans to the maintainable architectural solution (see Section 4).
- *Automated tooling*: Current approaches automatically capture ADDs (e.g. [33,35]). These approaches could be extended to support capturing DADDs. This requires extending current AK ontologies (e.g. [36]) with new concepts and terms regarding DADDs (e.g. ontology classes regarding DADDs' decision factors as explained in Section 4). Furthermore, automatic classification approaches (e.g. [35]) could be extended to capture the types of DADDs (as presented in Section 3).

7.1.2. Implications for practitioners

The detection of DADDs can allow re-using the knowledge (i.e. the rationale) behind making DADDs, and re-engineering the implementation of DADDs to maintainable architectural solutions. Moreover, capturing AK behind DADDs would also complement current methods of technical debt identification (e.g. capturing architectural technical debt [10]) by determining the rationale behind technical debt items. This would support software engineers to better prioritize and re-engineer technical debt items.

7.2. Making DADDs

Our empirical study shows that software engineers follow a systematic decision making process to decide on deliberate DADDs (see Section 4). Moreover, the results clarify the differences between deliberate and inadvertent DADDs (see Section 5), especially regarding their process, reasons and their impact on system quality (see Section 6).

7.2.1. Implications for practitioners

The results provide the first empirically-grounded decision making process for software engineers to make deliberate DADDs (see Section 4). The proposed decision making process is rather comprehensive, considering different factors as an integral part of design reasoning: project factors (e.g. project phase and resources), quality factors (e.g. level of technical debt) and human factors (e.g. stakeholder agreement) are considered to decide on deliberate DADDs (see Section 4). This process could be adopted by practitioners and organizations to ensure a systematic and reliable method for decision making on deliberate DADDs. Of course, it would need to be tailored to the needs of individual organizations, team and engineers.

The identified incidents and root causes of inadvertent DADDs in Section 5 could support companies to better manage technical debt caused by inadvertent DADDs. This could specifically impact the way companies gather requirements, as well as considerations to take when developing project and risk management plans.

7.2.2. Implications for researchers

The identified decision making process and decision factors can support the extension of current ADD management approaches and tools with additional concepts regarding DADDs. For example, Lytra and Zdun [37] proposed an ADD support framework and tools to guide software engineers during decision making. The approach provides a sequence of dependent questions and options. Based on the answers, the system recommends suitable architectural solutions. This approach could be extended with additional questions and options to consider decision factors to support making DADDs and re-engineering them. Moreover, the tool should allow selecting between maintainable and debt-incurring architectural solutions. This requires explicitly estimating the negative impact of a debt-incurring architectural solution, in terms of both the technical debt principal and interest [38].

8. Limitations and threats to validity

8.1. External validity

This aspect of validity is concerned with the generalizability of the results. One threat to the external validity comes from the fact that we did not select interviewees randomly, but contacted certain experts directly. This imposes limitations to how our results can be generalized to the various types of architects in different companies. Another limitation is the number of interviews conducted to obtain practitioner perspectives and viewpoints, which cannot guarantee *statistical* generalization. However, we followed two approaches to achieve *analytical* generalization [39]:

1. *We selected practitioners coming from different contexts*: We selected practitioners working in two different domains (i.e. embedded systems and enterprise applications). Moreover, we considered practitioners working in the same company, as well as practitioners working in different companies. Furthermore, practitioners work in companies with different sizes. This variety supported our analysis to determine most important concepts for DADDs.
2. *We achieved data saturation*: We did manage to achieve data saturation at the 9th interview. The last two interview confirmed previously identified concepts (in the other nine interviews, the focus group, and the follow-up questions), but did not lead to new concepts. This gave us confidence that we identified the most important concepts for DADDs.

8.2. Construct validity

This aspect of validity is concerned with the operational measures that were applied to answer the research questions (does the study investigates what it claims to be investigating?). One threat to the construct validity is related to the interview questions and their sensitivity, when asking about DADDs. For instance, some interviewees might not be willing to admit that they made wrong ADDs, or they may not want to expose issues created by others in the organization. To overcome this problem, we asked practitioners to provide anonymous scenarios (i.e. without mentioning the decision makers).

Moreover, in both the individual interviews and the focus groups, practitioners were asked to speak freely, and express their opinion without the researchers interrupting them. To support this, we followed common guidelines (e.g. [40]) for conducting interviews with practitioners.

Another threat to validity is the different implementation of the third step of data collection: having a focus group for the practitioners from the embedded domain versus sending follow-up questions to the practitioners from the enterprise domain. The difference resides in the ability of focus groups to initiate discussions between participants. We have partially mitigated this threat by having the same set of questions in both cases. Furthermore, combining the data from three different data collection methods (interviews, focus group, and follow-up questions), allowed for data triangulation, as we were able to confirm and refine the codes during data analysis. However, this threat cannot be fully mitigated.

8.3. Reliability

Reliability is concerned with the replicability of the study independently from the researchers, who conducted the study. One main threat to reliability is the bias of researchers and their expectations from the interviewees. To mitigate this threat, all authors have agreed upon a specific protocol for the interviews, as well as specific questions for the focus group and the follow-up questions. This ensured equal conditions among the different interviewees. We note that the follow-up questions were sent uniformly through email to the six practitioners from the enterprise domain; thus they could not be biased by individual researchers. Moreover, the interviews have been conducted by different researchers. For example, three of the interviews and the focus group were conducted by two authors of the paper, which have enriched the discussion with different questions and prevented steering the interview in a single direction.

To support replication of the study, as well as further future work, we make our interview questions available. During the data analysis of the interview transcripts, we evaluated the reliability of the identified concepts through reliability tests to ensure consensus and to support replication (see Section 2).

9. Related work

This section elaborates on related work on Technical Debt and Architectural Design Decisions in the following two sub-sections.

9.1. Technical debt

9.1.1. Evidence-based studies on technical debt

In the past decade, substantial research work on technical debt has been conducted. To highlight our contributions, we compare our results with those studies that are relevant to our work. We note that we only consider studies published in scientific literature and not grey literature (e.g. blog posts or white papers); as our work is evidence-based, we can only compare it with other evidence-based studies. Table 4 shows the intersection between our results and those studies. In the following paragraphs, we explain this intersection, organized per research question.

RQ1 - Types of DADDs. Our results in Section 3 extend the ontology of design decisions by Kruchten [11] with additional types of decisions, which incur technical debt. This new set of ADD types that are specific to technical debt, has not been previously reported in the literature. However, the fact that some design decisions produce certain types of architectural technical debt items, has been previously reported in the literature [8,24,42]. Nevertheless, these studies focus on the technical debt items, and do not define types of decisions.

RQ2 - Making deliberate DADDs. Our results in Section 4 provide the steps for making deliberate DADDs, based on empirical evidence from practitioners. These steps for making deliberate DADDs have not been previously reported in literature. However, some of the decision factors (from step 3) have been considered as causes of technical debt in current empirical studies (see Table 4). For instance, time constraints have been identified in several empirical studies as a cause of technical debt. Other decision factors on DADDs (e.g. priority of other disciplines over software or loyalty to company's products) have not been previously reported as causes of technical debt. Moreover, the agreement with developers and managers on DADDs (from step 4) have not been reported in current empirical studies on technical debt.

RQ3 - Triggers and root causes of inadvertent DADDs. Our results in Section 5 provide the first conceptual model (see Fig. 4), which clarifies how and why inadvertent DADDs happen. Several of the identified root causes and triggers of inadvertent DADDs have been previously reported as causes of technical debt (see Table 4). However, in addition to the conceptual model (Fig. 4), the relationships between root causes and triggers of inadvertent DADDs, have not been previously reported in the literature.

RQ4 - Dealing with DADDs. Our results in Section 6 explain how practitioners deal with existing DADDs, and which factors they consider to refactor DADDs. The current empirical studies on technical debt, have almost zero intersection with our results; most of the existing studies that study how to deal with technical debt (e.g. [44]) focus on proposing solutions (e.g. cost estimation techniques). In these solutions, the main decision factors are the characteristics of existing technical debt (e.g. principle and interest of technical debt). On the other hand, our results capture system, process and human factors, which influence the decision-making on dealing with DADDs.

9.1.2. Architectural technical debt

There has been some work on the nature and types of architectural debt. Martini and Bosch [8] identified 5 categories of architectural debt items based on an empirical study. Moreover, they determined their significant impact on the maintenance of the system. In another study [45], researchers investigated different architectural technical debt issues, which occur within a microservice architecture. In addition, they analysed their negative impact (i.e. interest) and possible solutions (i.e. principal). However, neither of these studies investigates the design reasoning behind the occurrence of architectural technical debt.

Several approaches identify architectural debt items by assessing system quality aspects. For example, Cai et al. [46] proposed a model and method to identify 'architectural roots', which are considered the main causes for maintainability issues. The method depends on capturing bug-prone files as a sign of critical architectural problems. Similarly, Fontana et al. [47] developed an approach to identify architectural technical debt through detecting architectural smells based on structural dependencies in source code. Xiao et al. [48] proposed an approach to identify architectural technical debt through capturing patterns of history changes and commits in a source code repository.

Li, Liang, and Avgeriou [49] proposed a process to capture architectural technical debt during architectural design; this allows preventing or tracking ATD before it is implemented in source code. Martini and Bosch [50] proposed a method to decide on refactoring architectural

Table 4
Relations between the results of this study and other empirical studies on technical debt.

	[8,24]	[9,41]	[42]	[43]	[3]
RQ1 — Types of DADDs					
Existence ADDs which incur requirements debt	X				
Existence ADDs which incur architectural debt	X		X		
Executive technology DADDs					
Build-versus-reuse technology DADDs			X		
RQ2 — Making deliberate DADDs					
Step 1: Determine maintainable architectural solution					
Step 2: Determine debt-incurring architectural solution					
Step 3: Analyse and tentatively select an architectural solution					
Decision factor - Time constraints	X	X	X	X	X
Decision factor - Availability of resources				X	
Decision factor - Complexity and quality aspects of the proposed architectural solutions				X	X
Decision factor - Level of technical debt in existing system	X				
Decision factor - Upcoming dependent requirements					
Decision factor - Re-usability of architectural solution	X				
Decision factor - Development phase					
Decision factor - Priority of other disciplines over software					
Decision factor - Loyalty to company's products					
Step 4: Agree and decide upon architectural design decision with stakeholders					
Agreement with software developers on DADDs					
Agreement with managers on DADDs					
Step 5: Document and plan maintainable architectural solution					
RQ3 — Triggers and root causes of inadvertent DADDs					
Inadvertent DADDs manifestation model (Fig. 4)					
Relationships between root causes and triggers of inadvertent DADDs					
Trigger - Introducing unexpected architecture-significant requirements		X	X	X	
Root cause - Wrong assumptions					
Root cause - Shallow architectural analysis due to agile and experimental approaches		X		X	
Root cause - Lack of architectural skills		X	X	X	X
Trigger - Discovering technology drawbacks, which hinder fulfilling ASRs		X			
Root cause - Lack of suitable technology alternatives					
Root cause - Software trends and sales over-claims					
Root cause - Lack of support from technology providers		X	X	X	X
Trigger - Changes in development team				X	
Root cause - Lack of documentation about design decision rationale		X	X	X	X
Root cause - Not considering knowledge and skills of other software engineers regarding architectural solutions				X	
RQ4 — Dealing with DADDs					
Availability of skilled resources					
Project phase		X			
Age of DADDs					
Importance of the project					
Risk of existing technical debt		X			
Company policy					
Complexity to identify and agree upon technical debt					

technical debt items. The method supports identifying factors involved in the growth of technical debt interest, and provides indicators for stakeholders to decide on technical debt refactoring. While these approaches propose useful tools for practitioners to identify architectural technical debt, they do not consider the reasoning of debt-incurring ADDs.

9.2. Requirements debt

Requirements debt was defined and discussed by Ernst [51] as the "distance between the optimal solution to a requirements problem and the actual solution, with respect to some decision space". We are not aware of any studies, which determine relationships between architectural design decisions and requirements debt, similarly to our classification in Section 3. However, current studies on requirements debt try to further define (e.g. [52]) and quantify requirements debt. For example, Abad and Ruhe [53] proposed a method to manage the uncertainty of requirements decisions using the previous history of a project.

9.3. Architectural design decisions

There is significant research work on architectural design reasoning and decision making [54]. However, currently there are no studies on the reasoning of debt-incurring ADDs in practice. In this section, we discuss relevant work on architectural design decisions and design reasoning.

Researchers proposed domain specific models and catalogues to guide software engineers, when taking ADDs in that domain. For example, Elmalki and Zdun [55] analysed common types of ADDs to guide decision making in service mesh based microservice architectures. Their results provide a model to guide software engineers during design space exploration. Malakuti et al. [56] proposed a catalogue for the types of ADDs when designing industrial IOT system. ADDs are associated with their decision factors from quality attributes as well as possible architectural solutions in the industrial IOT domain.

Design reasoning is a challenging physiological operation with many influences. For example, Zalewski et al. [57] identified twelve types of cognitive biases during architectural decision making. Tang et al. [58] proposed an approach to support software engineers with reasoning reminder card. The approach has been validated through experiments with practitioners and students. The result of the experiment shows that the reminder cards support software engineers to reason more on their decisions.

10. Conclusion and future work

Our study is motivated by the need to explore debt-incurring architectural design decisions in practice, either deliberate or inadvertent.

Specifically, we aimed at better understanding these types of design decisions, in order to extend current methods of architectural design decision making, and technical debt identification. To achieve this goal, we performed a case study, collecting data through user interviews and a focus group to allow for an in-depth investigation of practitioners' concrete experiences with DADDs.

Our results show that practitioners are very aware about the impact of debt-incurring ADDs. They have a process in place to decide between selecting a maintainable solution vs. a solution with technical debt. The decision is based on certain factors, and deliberations with a number of stakeholders. Moreover, practitioners deal with debt-incurring design decisions after implementation either pro-actively or re-actively.

We found differences in the impact and occurrence of deliberate and inadvertent debt-incurring ADDs. On the one hand, deliberate debt-incurring ADDs could be well planned and re-engineered later. On the other hand, inadvertent debt-incurring ADDs present a threat on the quality and architecture of a software system.

As future work, we plan to extend this study through interviewing and surveying additional practitioners regarding debt-incurring ADDs. For example, to generalize our results on other domains, we can investigate, if practitioners working in different domains reason similarly on DADDs. Moreover, we aim at extending current approaches of architectural knowledge capturing to detect debt-incurring ADDs. This would support two activities: capturing, sharing and re-using architectural knowledge; and assessing the quality of existing software systems.

CRediT authorship contribution statement

Mohamed Soliman: Conceptualization, Methodology, Validation, Investigation, Resources, Data curation, Writing - original draft, Writing - review & editing, Project administration. **Paris Avgeriou:** Funding acquisition, Conceptualization, Methodology, Investigation, Writing - review & editing, Project administration, Supervision. **Yikun Li:** Investigation, Data curation, Validation.

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.

Acknowledgements

We would like to thank ITEA3, The Netherlands and RV, The Netherlands for their support under grant agreement No. 17038 VISDOM (https://visdom-project.github.io/website). Moreover, we would like to thank the practitioners who volunteered to participate in our study.

References

[1] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, second ed., Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[2] P. Avgeriou, P. Kruchten, I. Ozkaya, C. Seaman, Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162), Technical Report 4, 2016, http://dx.doi.org/10.4230/DagRep.6.4.110, URL: http://drops.dagstuhl.de/opus/volltexte/2016/6693http://www.dagstuhl.de/16162.

[3] N.A. Ernst, S. Bellomo, I. Ozkaya, R.L. Nord, I. Gorton, Measure it? Manage it? Ignore it? Software practitioners and technical debt, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ACM, New York, NY, USA, 2015, pp. 50–60, http://dx.doi.org/10.1145/2786805.2786848, URL: http://doi.acm.org/10.1145/2786805.2786848.

[4] R. Kazman, Y. Cai, R. Mo, Q. Feng, L. Xiao, S. Haziyev, V. Fedak, A. Shapochka, A case study in locating the architectural roots of technical debt, in: Proceeding of International Conference on Software Engineering (ICSE), Vol. 2, IEEE Computer Society, 2015, pp. 179–188, http://dx.doi.org/10.1109/ICSE.2015.146.

[5] P. Avgeriou, N.A. Ernst, R.L. Nord, P. Kruchten, Technical debt: Broadening perspectives report on the seventh workshop on managing technical debt (MTD 2015), SIGSOFT Softw. Eng. Notes 41 (2) (2016) 38–41, http://dx.doi.org/10.1145/2894784.2894800, URL: http://doi.acm.org/10.1145/2894784.2894800.

[6] P. Kruchten, R.L. Nord, I. Ozkaya, Technical debt: From metaphor to theory and practice, 2012, http://dx.doi.org/10.1109/MS.2012.167.

[7] T. Besker, A. Martini, J. Bosch, Managing architectural technical debt: A unified model and systematic literature review, J. Syst. Softw. 135 (2018) 1–16, http://dx.doi.org/10.1016/j.jss.2017.09.025, URL: http://www.sciencedirect.com/science/article/pii/S0164121217302121.

[8] A. Martini, J. Bosch, The danger of architectural technical debt: Contagious debt and vicious circles, in: 2015 12th Working IEEE/IFIP Conference on Software Architecture, 2015, pp. 1–10, http://dx.doi.org/10.1109/WICSA.2015.31.

[9] A. Martini, J. Bosch, M. Chaudron, Architecture technical debt: Understanding causes and a qualitative model, in: Proceedings - 40th Euromicro Conference Series on Software Engineering and Advanced Applications, SEAA 2014, Institute of Electrical and Electronics Engineers Inc., 2014, pp. 85–92, http://dx.doi.org/10.1109/SEAA.2014.65.

[10] R. Verdecchia, I. Malavolta, P. Lago, Architectural technical debt identification: The research landscape, in: Proceedings - International Conference on Software Engineering, IEEE Computer Society, 2018, pp. 11–20, http://dx.doi.org/10.1145/3194164.3194176.

[11] P. Kruchten, P. Lago, H. Vliet, Building up and reasoning about architectural knowledge, in: C. Hofmeister, I. Crnkovic, R. Reussner (Eds.), Quality of Software Architectures, in: Lecture Notes in Computer Science, vol. 4214, Springer Berlin Heidelberg, 2006, pp. 43–58, http://dx.doi.org/10.1007/11921998{_}8.

[12] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, third ed., Addison-Wesley Professional, 2012.

[13] R. Capilla, A. Jansen, A. Tang, P. Avgeriou, M.A. Babar, 10 years of software architecture knowledge management, J. Syst. Softw. 116 (C) (2016) 191–205, http://dx.doi.org/10.1016/j.jss.2015.08.054.

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

[15] A. Tang, P. Avgeriou, A. Jansen, R. Capilla, M. Ali Babar, A comparative study of architecture knowledge management tools, J. Syst. Softw. 83 (3) (2010) 352–370, http://dx.doi.org/10.1016/j.jss.2009.08.032, URL: http://www.sciencedirect.com/science/article/B6V0N-4X4GHP5-1/2/84a45c0d6dda12f7f563273ff85be120.

[16] C. Manteuffel, P. Avgeriou, R. Hamberg, An exploratory case study on reusing architecture decisions in software-intensive system projects, J. Syst. Softw. 144 (2018) 60–83, http://dx.doi.org/10.1016/j.jss.2018.05.064.

[17] P. Runeson, Case Study Research in Software Engineering : Guidelines and Examples, Wiley, 2012, p. 237.

[18] S. Schröder, M. Soliman, M. Riebisch, Architecture enforcement concerns and activities - an expert study, J. Syst. Softw. 145 (2018) 79–97, http://dx.doi.org/10.1016/j.jss.2018.08.025.

[19] I. Seidman, Interviewing As Qualitative Research: A Guide for Researchers in Education and the Social Sciences, Teachers College Press, 2006.

[20] A. Strauss, J.M. Corbin, Basics of Qualitative Research: Grounded Theory Procedures and Techniques., Sage Publications, Inc, Thousand Oaks, CA, US, 1990, p. 270.

[21] J.C. Van Niekerk, J.D. Roode, Glaserian and straussian grounded theory: Similar or completely different?, in: ACM International Conference Proceeding Series, 2009, pp. 96–103, http://dx.doi.org/10.1145/1632149.1632163.

[22] K.J. Stol, P. Ralph, B. Fitzgerald, Grounded theory in software engineering research: A critical review and guidelines, in: Proceedings - International Conference on Software Engineering, Vol. 14-22-May-2016, IEEE Computer Society, 2016, pp. 120–131, http://dx.doi.org/10.1145/2884781.2884833.

[23] C. Miesbauer, R. Weinreich, Classification of design decisions: An expert survey in practice, in: Proceedings of ECSA 2013, Springer, 2013, pp. 130–145, http://dx.doi.org/10.1007/978-3-642-39031-9{_}12.

[24] A. Martini, J. Bosch, On the interest of architectural technical debt: Uncovering the contagious debt phenomenon, J. Softw.: Evol. Process. 29 (10) (2017) e1877, http://dx.doi.org/10.1002/smr.1877, URL: http://doi.wiley.com/10.1002/smr.1877.

[25] M. Soliman, M. Riebisch, U. Zdun, Enriching architecture knowledge with technology design decisions, in: WICSA, 2015, pp. 135–144, http://dx.doi.org/10.1109/WICSA.2015.14.

[26] F. Daneshgar, G.C. Low, L. Worasinchai, An investigation of 'build vs. buy' decision for software acquisition by small to medium enterprises, Inf. Softw. Technol. 55 (10) (2013) 1741–1750, http://dx.doi.org/10.1016/j.infsof.2013.03.009.

[27] A. Manjunath, M. Bhat, K. Shumaiev, A. Biesdorf, F. Matthes, Decision making and cognitive biases in designing software architectures, in: Proceedings - 2018 IEEE 15th International Conference on Software Architecture Companion, ICSA-C 2018, Institute of Electrical and Electronics Engineers Inc., 2018, pp. 52–55, http://dx.doi.org/10.1109/ICSA-C.2018.00022.

[28] V.S. Rekhav, H. Muccini, A study on group decision-making in software architecture, in: WICSA 2014 IEEE/IFIP, 2014, pp. 185–194, http://dx.doi.org/10.1109/WICSA.2014.15.

[29] D. Tofan, M. Galster, I. Lytra, P. Avgeriou, U. Zdun, M.A. Fouche, R. De Boer, F. Solms, Empirical evaluation of a process to increase consensus in group architectural decision making, Inf. Softw. Technol. 72 (2016) 31–47, http://dx.doi.org/10.1016/j.infsof.2015.12.002.

[30] C. Yang, P. Liang, P. Avgeriou, U. Eliasson, R. Heldal, P. Pelliccione, T. Bi, An industrial case study on an architectural assumption documentation framework, J. Syst. Softw. 134 (2017) 190–210, http://dx.doi.org/10.1016/j.jss.2017.09.007.

[31] U. van Heesch, P. Avgeriou, R. Hilliard, A documentation framework for architecture decisions., J. Syst. Softw. 85 (4) (2012) 795–820, http://dx.doi.org/10.1016/j.jss.2011.10.017.

[32] M. Soliman, M. Galster, A.R. Salama, M. Riebisch, Architectural knowledge for technology decisions in developer communities: An exploratory study with stackoverflow, in: IEEE/IFIP WICSA 2016, 2016, pp. 128–133, http://dx.doi.org/10.1109/WICSA.2016.13.

[33] A. Shahbazian, Y. Kyu Lee, D. Le, Y. Brun, N. Medvidovic, Recovering architectural design decisions, in: Proceedings - 2018 IEEE 15th International Conference on Software Architecture, ICSA 2018, Institute of Electrical and Electronics Engineers Inc., 2018, pp. 95–104, http://dx.doi.org/10.1109/ICSA.2018.00019.

[34] O. Zimmermann, J. Koehler, F. Leymann, R. Polley, N. Schuster, Managing architectural decision models with dependency relations, integrity constraints, and production rules, J. Syst. Softw. 82 (8) (2009) 1249–1267, http://dx.doi.org/10.1016/j.jss.2009.01.039.

[35] M. Bhat, K. Shumaiev, A. Biesdorf, U. Hohenstein, F. Matthes, Automatic extraction of design decisions from issue management systems: A machine learning based approach, in: Lecture Notes in Computer Science, in: (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Springer Verlag, 2017, pp. 138–154, http://dx.doi.org/10.1007/978-3-319-65831-5{_}10.

[36] M. Soliman, M. Galster, M. Riebisch, Developing an ontology for architecture knowledge from developer communities, in: IEEE/IFIP ICSA 2017, 2017, pp. 89–92, http://dx.doi.org/10.1109/ICSA.2017.31.

[37] I. Lytra, H. Tran, U. Zdun, Supporting consistency between architectural design decisions and component models through reusable architectural knowledge transformations, in: Software Architecture - 7th European Conference, ECSA 2013, Montpellier, France, July 1-5, 2013. Proceedings, in: Lecture Notes in Computer Science, vol.7957, Springer, 2013, pp. 224–239, http://dx.doi.org/10.1007/978-3-642-39031-9{_}20.

[38] A. Ampatzoglou, A. Ampatzoglou, A. Chatzigeorgiou, P. Avgeriou, The financial aspect of managing technical debt: A systematic literature review, in: Information and Software Technology, Vol. 64, Elsevier, 2015, pp. 52–73, http://dx.doi.org/10.1016/j.infsof.2015.04.001.

[39] W.A. Firestone, Alternative arguments for generalizing from data as applied to qualitative research, Educ. Res. 22 (4) (1993) 16–23, http://dx.doi.org/10.3102/0013189X022004016, URL: http://journals.sagepub.com/doi/10.3102/0013189X022004016.

[40] S.E. Hove, B. Anda, Experiences from conducting semi-structured interviews in empirical software engineering research, in: Software Metrics, 2005. 11th IEEE International Symposium, 2005, pp. 10 pp.–23, http://dx.doi.org/10.1109/METRICS.2005.24.

[41] A. Martini, J. Bosch, M. Chaudron, Investigating architectural technical debt accumulation and refactoring over time: A multiple-case study, in: Information and Software Technology, Vol. 67, Elsevier, 2015, pp. 237–253, http://dx.doi.org/10.1016/j.infsof.2015.07.005.

[42] R. Verdecchia, P. Kruchten, P. Lago, Architectural technical debt: A grounded theory, in: Lecture Notes in Computer Science, in: (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Springer Science and Business Media Deutschland GmbH, 2020, pp. 202–219, http://dx.doi.org/10.1007/978-3-030-58923-3{_}14, URL: https://link.springer.com/chapter/10.1007/978-3-030-58923-3_14.

[43] N. Rios, R.O. Spinola, M.G. De Mendonça Neto, C. Seaman, Supporting analysis of technical debt causes and effects with cross-company probabilistic cause-effect diagrams, in: Proceedings - 2019 IEEE/ACM International Conference on Technical Debt, TechDebt 2019, Institute of Electrical and Electronics Engineers Inc., 2019, pp. 3–12, http://dx.doi.org/10.1109/TechDebt.2019.00009, URL: https://ieeexplore.ieee.org/document/8785063/.

[44] C. Fernandez-Sanchez, J. Garbajosa, A. Yague, A framework to aid in decision making for technical debt management, in: 2015 IEEE 7th International Workshop on Managing Technical Debt, MTD 2015 - Proceedings, Institute of Electrical and Electronics Engineers Inc., 2015, pp. 69–76, http://dx.doi.org/10.1109/MTD.2015.7332628.

[45] S.S. De Toledo, A. Martini, A. Przybyszewska, D.I. Sjoberg, Architectural technical debt in microservices: A case study in a large company, in: Proceedings - 2019 IEEE/ACM International Conference on Technical Debt, TechDebt 2019, 2019, pp. 78–87, http://dx.doi.org/10.1109/TechDebt.2019.00026.

[46] Y. Cai, L. Xiao, R. Kazman, R. Mo, Q. Feng, Design rule spaces: A new model for representing and analyzing software architecture, IEEE Trans. Softw. Eng. 45 (7) (2019) 657–682, http://dx.doi.org/10.1109/TSE.2018.2797899.

[47] F.A. Fontana, I. Pigazzini, R. Roveda, M. Zanoni, Automatic detection of instability architectural smells, in: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2016, pp. 433–437, http://dx.doi.org/10.1109/ICSME.2016.33.

[48] L. Xiao, Y. Cai, R. Kazman, R. Mo, Q. Feng, Identifying and quantifying architectural debt, in: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), 2016, pp. 488–498, http://dx.doi.org/10.1145/2884781.2884822.

[49] Z. Li, P. Liang, P. Avgeriou, Architectural technical debt identification based on architecture decisions and change scenarios, in: 2015 12th Working IEEE/IFIP Conference on Software Architecture, 2015, pp. 65–74, http://dx.doi.org/10.1109/WICSA.2015.19.

[50] A. Martini, J. Bosch, An empirically developed method to aid decisions on architectural technical debt refactoring: Anacondebt, in: 2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C), 2016, pp. 31–40, http://dx.doi.org/10.1145/2889160.2889224.

[51] N.A. Ernst, On the role of requirements in understanding and managing technical debt, in: 2012 3rd International Workshop on Managing Technical Debt, MTD 2012 - Proceedings, 2012, pp. 61–64, http://dx.doi.org/10.1109/MTD.2012.6226002.

[52] V. Lenarduzzi, D. Fucci, Towards a holistic definition of requirements debt, in: International Symposium on Empirical Software Engineering and Measurement, Vol. 2019-Septemer, IEEE Computer Society, 2019, http://dx.doi.org/10.1109/ESEM.2019.8870159.

[53] Z.S.H. Abad, G. Ruhe, Using real options to manage technical debt in requirements engineering, in: 2015 IEEE 23rd International Requirements Engineering Conference, RE 2015 - Proceedings, Institute of Electrical and Electronics Engineers Inc., 2015, pp. 230–235, http://dx.doi.org/10.1109/RE.2015.7320428.

[54] M. Razavian, B. Paech, A. Tang, Empirical research for software architecture decision making: An analysis, J. Syst. Softw. 149 (2019) 360–381, http://dx.doi.org/10.1016/j.jss.2018.12.003.

[55] A. El Malki, U. Zdun, Guiding architectural decision making on service mesh based microservice architectures, in: Lecture Notes in Computer Science, in: (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Springer Verlag, 2019, pp. 3–19, http://dx.doi.org/10.1007/978-3-030-29983-5{_}1.

[56] S. Malakuti, T. Goldschmidt, H. Koziolek, A catalogue of architectural decisions for designing iIoT systems, in: Lecture Notes in Computer Science, in: (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Springer Verlag, 2018, pp. 103–111, http://dx.doi.org/10.1007/978-3-030-00761-4{_}7.

[57] A. Zalewski, K. Borowa, A. Ratkowski, On cognitive biases in architecture decision making, in: Lecture Notes in Computer Science, in: (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Springer Verlag, 2017, pp. 123–137, http://dx.doi.org/10.1007/978-3-319-65831-5{_}9.

[58] A. Tang, F. Bex, C. Schriek, J.M.E. van der Werf, Improving software design reasoning–a reminder card approach, J. Syst. Softw. 144 (2018) 22–40, http://dx.doi.org/10.1016/j.jss.2018.05.019.