Comes later manuscript No.
(will be inserted by the editor)

On designing a process for identifying Architectural Technical Debt from Bug Reports

Lakmal Silva · Michael Unterkalmsteiner · Krzysztof Wnuk

Received: date / Accepted: date

Abstract ***Background:*** A key challenge with Architectural Technical Debt (ATD) is ATD identification. Most existing methods use source code analysis that does not capture the hidden ATD in the architecture design. A few studies focus on ATD identification in software architecture design, but it is unclear whether they can scale to industrial contexts.

Objectives: We design and illustrate the evolution of an efficient ATD identification process using bug reports. Furthermore, we validate the process by analyzing actual bug reports filed by customers.

Method: Design Science Research was used to design and evolve an ATD identification process in three design iterations. The iterations were conducted at Ericsson, focusing on improving the process to scale in large software-intensive systems.

Results: We report how we engineered the ATD identification process. In particular, we illustrate the results from each design iteration, discuss the lessons learnt, and demonstrate the implementation of the final ATD identification process for decision-making activities at Ericsson. We identified 30 ATD related defects from a random sample of 251 bug reports that spanned over four releases of the system we investigated.

Conclusions: Our study indicates that bug reports can be used as a data source for identifying ATD using our proposed process and decision-making activities. Furthermore, we emphasize the need for clarity and consensus among the stakeholders regarding the scope and boundaries of architectural components that constitute a software system for the ATD analysis to be relevant.

Keywords Architectural Technical Debt · ATD · bug reports · System Evolution · Orthogonal Defect Classification · C4model

Lakmal Silva
Ericsson AB, Sweden and Blekinge Institute of Technology, Sweden. E-mail: lakmal.silva@bth.se

Michael Unterkalmsteiner
Blekinge Institute of Technology, Sweden. E-mail: michael.unterkalmsteiner@bth.se

Krzysztof Wnuk
Blekinge Institute of Technology, Sweden. E-mail: krzysztof.wnuk@bth.se

1 Introduction

Software development involves delivering products within the constraints of budgets and time. This leads to compromises during product design and implementation caused by limited resources. Technical Debt (TD) is a term coined by Cunningham (1992) referring to taking sub-optimal compromises. The accumulation of such compromises hinders the evolution and maintenance of a system in the long run (Besker et al. 2018). TD can incur in different forms such as the widely explored source code-related TD, requirements TD, testing TD, and design TD. Among these TD types, the design decisions taken on the software architecture are a key source of TD (Ernst et al. 2015). The TD associated with the sub-optimal software architecture design decisions (structure, technologies, programming languages, development processes, platforms) is referred as Architectural Technical Debt (ATD) (Verdecchia et al. 2021).

According to the unified model of ATD proposed by Besker et al. (2018), the ATD Management (ATDM) process constitutes of multiple phases: identification, measurement, prioritization, repayment, and monitoring of ATD. ATD identification remains a challenging task (Besker et al. 2018; Verdecchia et al. 2018; Li et al. 2015). There is a shortage of ATD identification techniques in relation to the architecture design decisions, structure, and technology choices (Verdecchia et al. 2018; Besker et al. 2018). Most of the existing ATD identification techniques are based on source code analysis (Garcia et al. 2009; Xiao et al. 2021) and do not capture deficiencies related to the architectural design aspects of the system. Source code analysis detection techniques focus on the implemented system rather than the design of the system, where critical architecture decisions are made. For example, it is difficult to understand the implications of choosing a library or a framework, entirely by performing source code analysis. These sub-optimal architecture designs are more challenging to detect, hence more severe as they may remain undetected for a longer period. Moreover, code analysis techniques lack mechanisms to confirm whether the identified ATD is indeed real ATD (Li et al. 2015). Software architecture and ATD is in the invisible spectrum of the software technical debt landscape (Kruchten et al. 2012). The invisible nature of ATD is a challenge for the ATD detection (Kruchten et al. 2012) that is buried in the decision being made at early design phases. The lack of attention to ATD and its treatment may result in high maintenance costs and hinder new feature introduction (Xiao et al. 2016; Besker et al. 2018).

On the other hand, software defects are in the visible spectrum (Kruchten et al. 2012) of the technical debt landscape. The discovered defects, reported as bug reports, are essential artifacts generated throughout the life cycle of a software system. Bug reports are generated during the software development process and during operations. Mining bug report repositories has been a well-researched area both in industry and research communities for several decades. Processes such as the Fault Slip Through (FST) (Damm et al. 2006) were introduced to detect defects as early as possible in the software development process. However, our experience with Ericsson revealed that many defects are being discovered late in the development process and in operations. This has led to a costly software maintenance process (Dehaghani and Hajrahimi 2013) due to fixing and delivering of defects. What if these defects are caused by or related to invisible ATD? If we can detect ATD related defects, we could treat ATD by redesign the system components that accumulated ATD. These

redesigning efforts would be paid off in the long run due to the reduced number of defects.

Our study takes a pragmatic approach to identifying ATD by utilizing bug reports as the primary data source. As pointed out by Verdecchia et al. (2021), certain defects may be an indication of the presence of ATD. We designed an ATD identification process in an industrial context together with a product development unit at Ericsson. We used design science research to design and improve the ATD detection process by obtaining feedback from practitioners at Ericsson in each iteration.

The **main contribution** of this study is a process to systematically analyse bug reports with the goal of identifying ATD. From an industrial standpoint, we believe that bug reports are a useful data source and make the process more pragmatic and easier to integrate into existing design maintenance routines such as FST. The lightweight nature of the process makes it more suitable to be used on a regular basis as part of continuous improvements in Agile processes. From a research standpoint, we believe our approach provides a novel direction for ATD identification using bug reports as a data source, which has not been the focus of ATD research. Another contribution is lessons learnt from the journey of building the process such as the need for visualizing and defining the boundaries or scope of different architectural components.

The rest of the paper is structured as follows. Section 2 provides related work in the context of ATD identification. Then we detail our research methodology and introduce the research questions in Section 3. In Section 4, we describe our approach and report the results from the three design iterations. The research questions that are answerable by the results are highlighted within the results. The industrial empirical implementation of our ATD identification process and the lessons learnt are reported in Section 5. We compare our ATD identification process to similar approaches from prior research in Section 6. We conclude the paper and highlight the future directions in Section 7.

2 Related Work

This study focuses on ATD identification from bug reports. Bug analysis is a well-researched area where a majority of bug analysis studies focused on bug triage (the process of assigning a bug report to a developer), duplicate bug report detection (Wu et al. 2011), defect prediction (Chang et al. 2009), to name a few. However, there is little focus on defect analysis from an architectural perspective.

Most proposed ATD identification methods are based on source code analysis (Verdecchia et al. 2018; Li et al. 2015). Information about decisions relevant to the architecture is also captured beyond the source code and often scattered in multiple artifacts such as documents, chat logs and emails (Pérez et al. 2019).

We summarize different types of ATD identification methods from prior research in the remainder of this section to provide the current state of the art.

ATD issues pertaining to microservice-based architectures were identified by de Toledo et al. (2021) in their study involving interviews with 25 practitioners. The identified ATD issues can be thought of as ATD categories according to the terminology used by Verdecchia et al. (2018). These categories can be used when identifying ATD from bug reports in microservices based systems.

Xiao et al. (2021) present four typical patterns of ATD and identify ATDs that persistently incur significant maintenane costs. They introduce a trajectory of the maintenance costs model for ATD. The automated detection of the ATD locations is helpful for maintenance cost prediction. This could support software organizations to prioritise the improvement areas within the set of grouped files.

A machine learning approach was employed by BenIdris et al. (2020) in their study of identifying and prioritizing ATD. The approach involves utilizing internal structure metrics related to component cohesion, number of methods associated with a component and modularization. The severity of ATD for components was determined based on the calculated values of the internal structure metrics. The results are however based on projects written in the C# programming language, so it is uncertain if the method can be utilized for projects written in other languages.

Martini et al. (2018a) used an existing tool "Arcan" (Fontana et al. 2017) for automatically detecting dependency related (unstable, hub-like and cyclic dependencies) architectural smells in Java projects, followed by an assessment of the findings with practitioners. They concluded that architectural smells are useful for identifying and prioritising ATD. However, the conclusions are based on a single type of ATD related to dependencies.

With a specific focus on non-modularity, Martini et al. introduced a measurement system for identifying ATD according to stakeholder goals (Martini et al. 2018b). The lack of modularity was calculated based on files of interest (FOI) in the context of stakeholder goals. The presence of complex FOI connected to functionality that often changes were considered as a symptom of ATD.

The ATD identification method introduced by Xiao et al. (2016) also uses the idea of grouping architecturally connected files together with a debt model to represent the interest rate of the debts. The study is based on four types of ATDs (unstable interfaces, modularity violations, unhealthy inheritance and cyclic dependency).

A different approach to ATD identification was taken by Li et al. (2015), where they utilized architecture decisions and change scenarios as opposed to source code analysis. However, a downside of this approach is that it requires keeping track of architecture decisions and changes over time to determine if ATD is accumulating.

An important observation we made from prior studies is that the interpretation of ATD varies in different studies, as summarized in Table 1. Based on this summary, we see two distinct interpretations of the architecture. One interpretation is based on defining different source code metrics, files, and structures in relation to architecture (Xiao et al. 2021; BenIdris et al. 2020; Martini et al. 2018a,b; Xiao et al. 2016). The second interpretation is based on architecture decisions (de Toledo et al. 2021; Li et al. 2015). This is similar to the two kinds of ATD that Pérez et al. (2019) acknowledged in their study: "Solution ATD" and "Decision ATD". Decision ATD refers to the ATD incurred during the decision making processes mostly in the design phases whereas Solution ATD is related to the ATD incurred during the implementation phases that involves the source code.

ATD identification methods are based on an individual's interpretation of ATD. The term "architecture" is an overloaded term, as we point out in Section 4.3.2. Hence, being a derivative of the architecture, ATD also suffers from the term overloading effect. Therefore, we emphasize the need to define the level and the scope of the architecture abstractions used in ATD studies to clearly distinguish ATD from the generic forms of technical debt.

Table 1 ATD identification strategies

Paper Title	Method	Interpretation of ATD
Identifying architectural technical debt, principal, and interest in microservices: A multiple-case study (de Toledo et al. 2021)	Interviews and open coding based categorization	sub-optimal architectural solutions
Detecting the Locations and Predicting the Maintenance Costs of Compound Architectural Debts (Xiao et al. 2021)	Source code mining to automatically detect ATD locations	"A group of architecturally connected files that persistently incur high maintenance costs over time" (Xiao et al. 2021)
Prioritizing Software Components Risk: Towards a Machine Learning-based Approach (BenIdris et al. 2020)	Used internal structure metrics to identify and classify ATD	code smell analysis through machine learning
Identifying and Prioritizing Architectural Debt Through Architectural Smells (Martini et al. 2018a)	Architectural smell detection through Arcan tool (Fontana et al. 2017) and use of questionnaires, interviews	Architecture Smells as a relation to ATD
A semi-automated framework for the identification and estimation of architectural technical debt: A comparative case-study on the modularization of a software component (Martini et al. 2018b)	Identify non-modular components based on source code quality parameters	The presence of complex FOI
Identifying and Quantifying Architectural Debt (Xiao et al. 2016)	source code analysis to detect architecturally connected files	A group of architecturally connected files and their maintenance cost growth
Architectural technical debt identification based on architecture decisions and change scenarios (Li et al. 2015)	Using Architecture Decision Records together with decision based and scenario based scenarios through interviews	Architecture Decisions that compromises quality attributes

3 Research Methodology

Our conjecture is that Architectural Technical Debt (ATD) has an observable effect on the perceived quality of a software product or service. Bug reports are an important quality attribute of a software system. Thus, our objective is to design a process that facilitates ATD identification using bug reports as a data source. We used Design Science Research to design and evolve our ATD identification process, as outlined in Figure 1 and explained in the remainder of this section.

To guide the process design, we formulated the following research questions (RQs):

RQ1 What ATD categories exist in the literature?
RQ2 To what degree can the ATD categories defined in the literature support the identification of ATD from bug reports?
RQ3 Which of the identified ATD categories can we observe in a large software-intensive system?

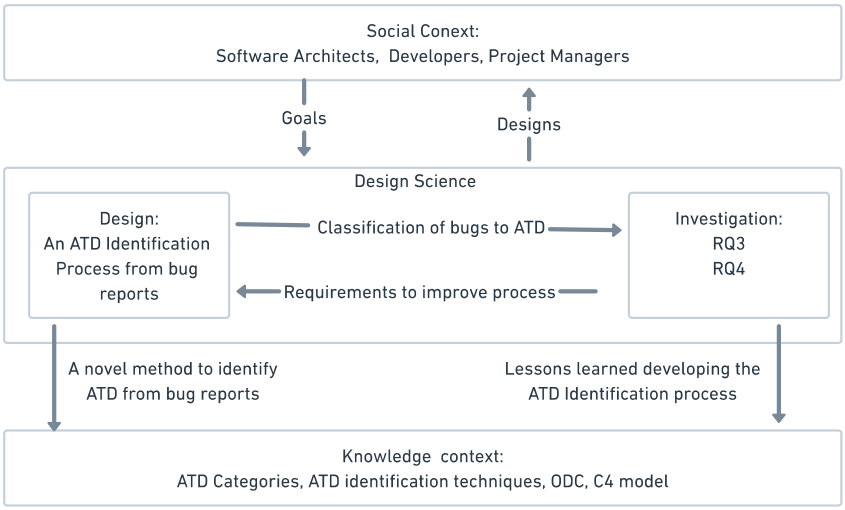

Fig. 1 Design Science Research framework adapted from Wieringa Wieringa (2014)

RQ4 What are the lessons learned from designing an ATD identification process from bug reports?

The motivation behind RQ1 is to consolidate state-of-the-art ATD knowledge gathered from prior research in the context of ATD categories. We used a literature survey that included both Systematic Literature Reviews (SLRs) and Systematic Mapping Studies (SMSs) to answer this RQ.

We assess whether the identified ATD categories are appropriate for ATD identification by answering RQ2. We conduct an interview based case study at a product development unit in Ericsson involving software architects and senior developers for this purpose.

Through RQ3, we merge the academic knowledge of ATD categories with an industrial data set of bug reports to explore the effectiveness and accuracy of identifying ATD categories.

To contribute with knowledge that can be utilized in future research (Patton 2001) related to ATD identification from bug reports, we introduced RQ4 to share lessons we learnt from our research.

The research comprises three design iterations in which we improve the classification process with each iteration. Figure 2 depicts the iterations. The classification process was implemented within a development unit at Ericsson that develops a software-intensive system, hereafter referred to as "System A". Ericsson personnel was involved in iteration two and iteration three. Apart from the participants listed in Table 2, the lead author, who is the main driver of this research, is employed by Ericsson for the past 14 years and working in the capacity of a Software Architect.

To bootstrap the research, we identified existing ATD categories from prior research by performing a literature survey of existing systematic literature and mapping studies (RQ1). Then, we evaluated the identified ATD categories and definitions for their ATD identification capability in an industrial software-intensive system (RQ2). We conducted interviews with the architecture team members of System A

Table 2 Participant Demographics

Id	Ericsson Role	Education	Overall experience	Experience at Ericsson	Involved iteration
P1	Chief Architect	Licentiate	23	23	1, 2, 3
P2	Software Architect	Masters	28	28	2
P3	Software Architect	Masters	26	23	2
P4	Senior Developer	Masters	17	17	2
P5	Senior Developer	Masters	21	21	2
P6	Developer	Bachelor	10	2	3
P7	Developer	Bachelor	7	6	3

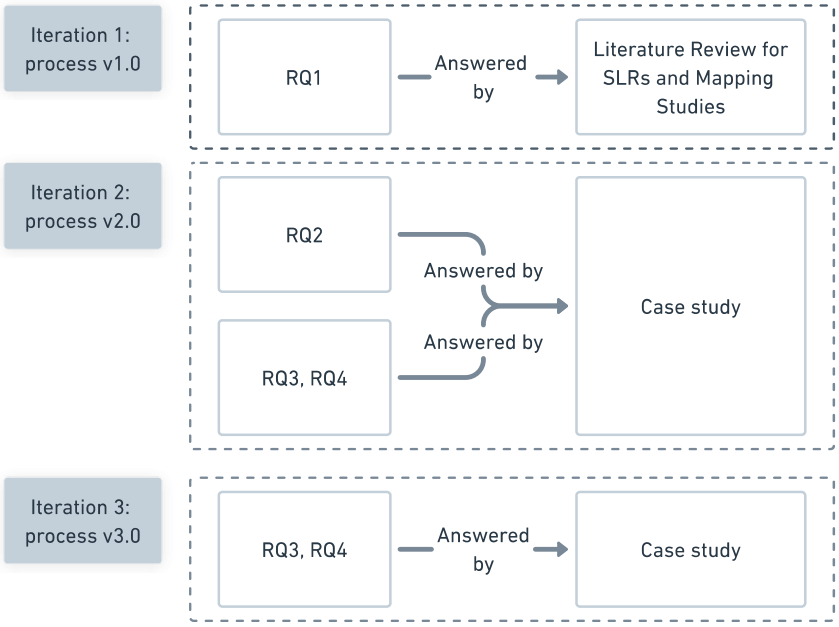

Fig. 2 Research Questions and Research Methods

and investigated Generic Software Architecture Guidelines at Ericsson. The ATD identification process needs to be efficient for it to be adapted in industry; hence we improved the process in each design iteration by mitigating the challenges and eliciting the lessons learnt from the previous design iterations (RQ4). We classified 251 bug reports, determining which ATD categories were found in System A (RQ3) and indicating which ATD categories were found to be useful in practice.

4 Design iterations and Results

Figure 3 depicts the three design iterations and their detailed steps. Each step is prefixed with I<digit>, A<digit>, and O<digit> to indicate the Inputs to the iterations, Actions performed within the iterations, and the Outputs from the iterations, respectively. Specific Outputs of one iteration become an Input to the next iteration.

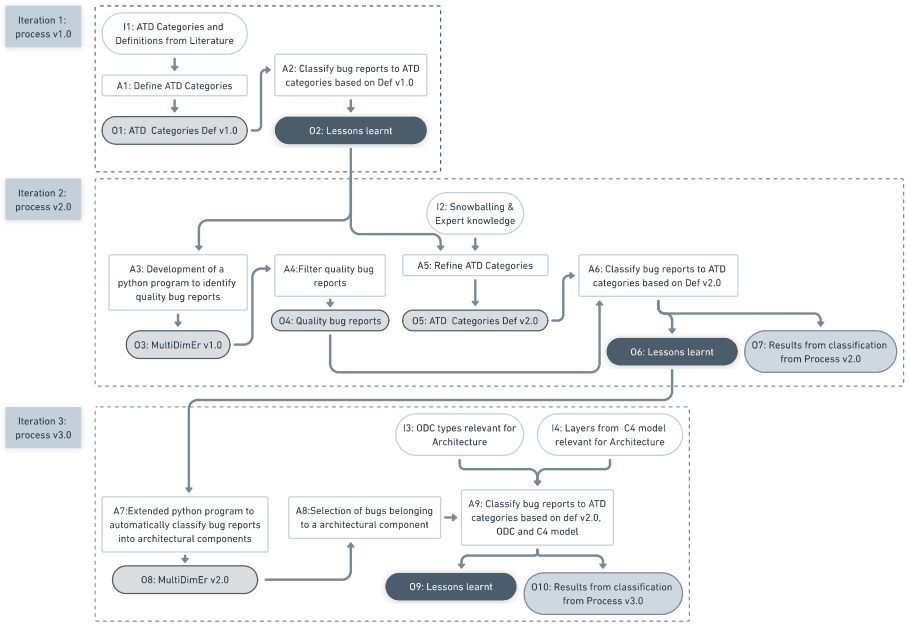

Fig. 3 Inputs, Outputs and Actions in Design Iterations

For example, the lessons learnt from iteration 1 (O2: Lessons learnt) becomes input to iteration 2 to improve the process. The ATD identification derived in each of the three iterations are labelled with a version, process v1.0, process v2.0 and process v3.0 for ease of referencing.

The remainder of this section describes the approach used and reports the results from each design iteration. Following the guidelines for conducting and reporting case study research provided by Runeson and Höst (2009), we provide the process details as a chain of evidence to follow the derivations of results. The content is structured in chronological order. The selected research approach and results of each iteration are reported together under each iteration. The results from each iteration are the outputs which are prefixed as O<digit>. The results also answer the research questions RQ1-RQ4.

4.1 Iteration 1

The goal of ATD category identification is to support various stakeholders (e.g. Software Architect or a Senior designer) in matching the bug reports with ATD categories. The lead author conducted a literature review to identify ATD categories. The following search string was used in Scopus[1] to find relevant SLRs and or SMSs related to ATD identification. The set of papers are input in "Iteration 1", shown as I1 in Figure 3.

[1] `https://www.scopus.com/`

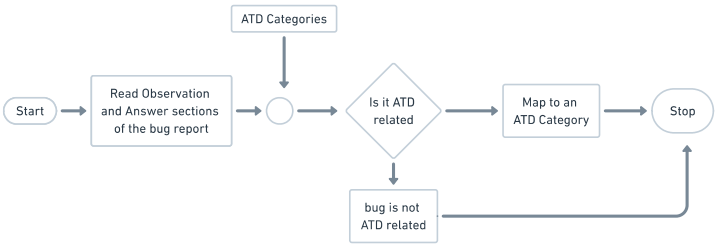

Fig. 4 ATD identification process used in iteration 1 and Iteration 2

TITLE-ABS-KEY (software AND ("ATD" OR "architect* technical debt") AND (review OR slr OR sms OR mapping OR survey)) AND (LIMIT-TO (SUBJAREA,"COMP") OR LIMIT-TO (SUBJAREA,"ENGI"))*

In the next step (A1: Define ATD Categories), we identified two baseline papers, an SLR (Besker et al. 2018) and a Systematic Mapping study (Verdecchia et al. 2018) based on the criteria that they are related to ATD identification. We extracted four main ATD categories (O1: ATD Categories Def v1.0) from the the two papers: dependency violations, non-modularity, compliance violations, and change proneness. These four categories answer RQ1.

RQ1 What, ATD categories exist in literature?
ATD Categories Def v1.0

- dependency violations
- non-modularity
- compliance violations
- change proneness

We conducted a pilot study where we categorized 45 bug reports from System A to the identified ATD categories and definitions (A2: Classify bug reports to ATD categories based on Def v1.0) according to the process outlined in Figure 4. We experienced several ambiguities among the categories. An example of such a definition is the ATD category "dependency violations": Architectural violations resulting from unfit dependencies among architectural components (Verdecchia et al. 2018). Our assumption was that the observation and answer section of a bug report may have documented evidence of such architectural debt.

As illustrated in Figure 4, the ATD classification process starts by selecting a bug report and reading the Observation and the Answer sections to understand the defect context and the cause of the defect. Then, based on the expert knowledge of the system, the engineer determines if the defect is due to architectural issues. If it is architecture-related, the engineer attempts to map the defect to one or more of the four ATD categories.

However, we realized that the abstract nature of the four categories and their definitions identified from prior research were insufficient for bug classification. For instance, we found that the ATD category "compliance violations" is too abstract to be used in our classification without the types of compliance being explicitly defined in the system context. Additionally, these definitions need to be not overlapping with

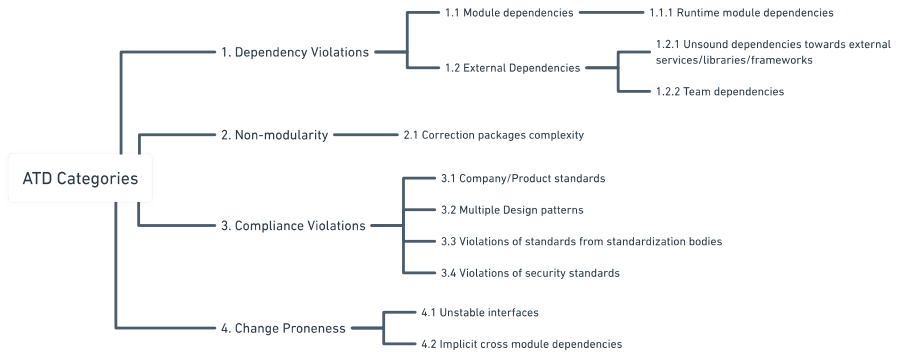

Fig. 5 ATD Categories Def v2.0

one another to minimize classification ambiguity. We can summarize the two lessons learnt (O2: Lessons learnt) from the pilot classification:

Lesson1 ATD categories should be anchored in the context of the software system under consideration.

Lesson2 ATD categories should not significantly overlap with one another.

> *RQ2 To what degree can the ATD categories and definitions found in the literature support the identification of ATD from bug reports?*
> We observed that the ATD categories from literature provided us with a foundation for defect mapping. However, the descriptions were not concrete enough for our classification purposes.

4.2 Iteration 2

The lessons learnt (O2: Lessons learnt) from iteration 1 were used as input to refine the ATD categories. We identified more concrete and fine-grained ATD category definitions by back- and forward snowball sampling (I2: Snowballing & Expert knowledge) from the initial two papers (Besker et al. (2018) and Mumtaz et al. (2020)) from which the definitions originated. Furthermore, we conducted interviews with the members of the architecture team of System A and also reviewed the Generic Architecture Guidelines that Ericsson systems need to comply with. This strategy helped us to extract specific ATD categories in the context of large software-intensive systems (A5: Refine ATD Categories). The improved list of ATD categories is shown in Figure 5 and the definitions are outlined in Table 3 (O5: ATD Categories Def 2.0).

Using the improved ATD category definitions, we classified another sample of 45 bug reports from System A, following the process outlined in Figure 4, and in Section 4.1. The second pilot revealed that the observation and answer sections in bug reports are occasionally written poorly, referring to email discussions that could not be traced and one-line answers with source code patch identifiers. We learnt the following lesson from this exercise:

Table 3: ATD Categories Def v2.0 and Description

Index	ATD Category	Source	Description
1	Dependency violations	Verdecchia et al. (2018)	"Architectural violations resulting from unfit dependencies among architectural components. Such types of Faults are usually caused by unsound architectural design choices, incorrect implementation, or architectural deterioration. The presence of architectural dependencies (for example at different component/module levels) which are considered forbidden in the (context-specific) architecture" (Verdecchia et al. 2018)
1.1	Module dependencies	Besker et al. (2018)	The behavior of one element influences the behavior of another element(module), e.g., the responsibility of one element changes, resulting in other elements needing to change their architectural role. Example: A module in this particular Ericsson context could be a service running in a single Java Virtual Machine (JVM) instance
1.1.1	Runtime module dependencies	system/ industrial context	A component that, when executed, should not trigger the execution of another component, as specified by the architects/architecture. Example: Startup of modules should not have dependencies to one another. If a service is not available, The module should start and wait for the availability of the required service
1.2	External dependencies	Besker et al. (2018)	Interface or contract-based interactions between elements realized through a connector
1.2.1	Unsound Dependencies towards external services/libraries/frameworks	system/ industrial context	Quality attributes of one element constrain other elements, e.g., license, technology choice, or usage patterns. An element utilizes another element 'owned' or managed by another system which was not meant to be a shareable dependency. Lack of alignment to interfaces and methods when updating 3pps, Faults due to non-optimally selected 3rd party libraries/frameworks, Chains of dependencies not fit for purpose Example: Selection of proper licenses that don't expose Ericsson source code. Wrapping of 3pps within a container and expose
1.2.2	External Team dependencies	Besker et al. (2018)	Due to economic and strategic reasons, it is common to setup external teams in addition to core teams that are driving system development. If not managed properly, this can result in an unforeseen selection of 3rd party libraries, architecture patterns which results in deteriorating the system (Silva et al. 2018).

2	Non-modularity	Verdecchia et al. (2018)	"Sub-optimal modularization of architectural components. Lack of modularity often causes small changes to propagate to other portions of a system, lowering the maintainability and evolvability of software systems. Can affect architectural requirements (performance, robustness, among others)" (Verdecchia et al. 2018). Modularity is as essential characteristic of modern software system design for "improving the flexibility and comprehensibility of a system while allowing the shortening of its development" (Parnas 1972). Symptoms such as the difficulty to change a database without having to change bulk of the other modules of the system, inability to delivery only the impacted modules are few examples of non-modularity or sub optimal modularization of the system. Parnas in his papar on "On the Criteria To Be Used in Decomposing Systems into Modules" (Parnas 1972) suggested to start decomposing a system such that hard decision decisions and the decisions that are likely to change, to be hidden from one another. This will minimize the symptoms we mentioned earlier.
2.1	Correction packages complexity	system/ industrial context	The monolithic nature of the system forces to delivery of large parts of the system that makes patch updates more complex, error-prone and time consuming. It can also contribute the accidental delivery of unintended parts of the system that can create additional issues.
3	Compliance Violations	Verdecchia et al. (2018)	"Deviation with respect to a certain architectural pattern (e.g. model-view-controller) affecting the quality of the system" (Verdecchia et al. 2018). This category comprehends patterns and policies that are not kept consistent throughout the system. Not using proper naming conventions with respect to the domains and the program languages in a certain context, non-uniform design or architectural patterns being used to solve the same problem in different parts of the system are a few examples that can be considered as compliance violations.
3.1	Company/Product rule violations	system/ industrial context	Most software development companies has generic product guidelines that products need to comply with. Additionally the architects of System A defined certain design principles such as module independence, location transparency (i.e., the service implementation should not be aware of how they are wired in a networked environment) that should be met when implementing System A.

3.2	Multiple design patterns	system/ industrial context	The presence of different design or architectural patterns used to implement the same functionality.
3.3	Violations of standards from standardization bodies	system/ industrial context (Silva et al. 2018)	Non-Compliant to standards issued by different standardization bodies such as the Engineering Task Force (IETF) [2], The World Wide Web Consortium (W3C) [3] and The 3rd Generation Partnership Project (3GPP) [4] to name a few. In some situations, non-compliance results in losing business.
3.4	Violations of security standards	system/ industrial context	Violation or not adhering to security principles and security best practices such as the Open Web Application Security Project (OWASP) top ten [5] and Common Weakness Enumeration (CWE) top 25 [6].
4	Change Proneness	Verdecchia et al. (2018)	"Refers to architectural components that are modified with a high frequency" (Verdecchia et al. 2018). For example, frequent changes of architectural components such as interfaces, core modules that other parts of the system are relying on, makes the overall system more error-prone and unstable. Such key architectural components need to be designed properly when laying the architecture foundations of the system.
4.1	Unstable Interfaces	system/ industrial context	An unstable interface is a leading file with a large number of dependents but changes frequently, which results in changing many other dependency files. An interface is a contract between components/modules. An unstable interface can be error-prone and can introduce compatibility issues. If such an interface is an external interface, which is used to interact with an external system, issues may not even be detected until a newer version of the system being deployed at a customer premises. It's also costly to make simple changes due to the large number of files that require updates
4.2	Implicit cross module dependency	system/ industrial context	Files belong to different independent modules but are changed together frequently. This may be a result of non-optimal modularization which might have resulted in duplicated code or that the system decomposition criteria is not proper.

Lesson3 Bug reports should contain rich information in terms of describing the issue, problem analysis, and the implemented solution to be usable for ATD identification.

[2] https://www.ietf.org/

[3] https://www.w3.org/

[4] https://www.3gpp.org/

[5] https://owasp.org/www-project-top-ten/

[6] https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html

To explore the issues with bug descriptions, we needed to evaluate the quality of bug reports. We made an assumption that the number of sentences in a bug report indicates how descriptive the content is. In other words, more sentences would mean that there is a better analysis of the defect that can provide helpful information required for the ADT classification. We developed a tool (A3: Development of a python program to identify quality bug reports) using the Natural Language Tool Kit (NLTK[7]) for counting the number of sentences in the answer section as well as the defect description section. We sorted the bug reports in descending order, first by the number of sentences in the answer section and then by the sentences in the defect description. This defect description analysis tool was the first version of a bug report analysis tool which we further developed in the next iteration. We called it the Multi-Dimensional bug Analyzer (MultiDimEr) (Silva et al. 2022) when we further enhanced it for bug report analysis in more dimensions such as bug distribution over architectural components, severity, bug report answer codes, among others. O3: MultiDimEr v1.0 in iteration 2 helped us to identify quality bug reports (A4:Filter quality bug reports) to be analyzed by our process.

4.2.1 Industrial Case Study - Iteration 2

To validate the ATD classification process v2.0, we conducted an industrial case study at a product development unit in Ericsson to classify bug reports to ATD categories where relevant by interviewing five senior staff members that have been working with System A.

Three Architects and two senior developers (P1 to P5 in Table 2) were selected to validate the ATD classification process v2.0. They have been working with System A for over ten years. A joint information session was conducted with all five participants and two authors of this paper. As suggested by Runeson and Höst (2009), we prepared a letter of consent which was signed by all participants before proceeding with the session. The motivations behind the defect classification, different ATD categories, and the defect classification process were introduced during this session. The participants were given time to read through the ADT categories/subcategories definitions (O5: ATD Categories Def v2.0) and to ask clarification questions.

After this joint session, each participant was invited individually for a one-hour data extraction session (A6: Classify bug reports to ATD categories based on Def v2.0) together with the authors. The goal was that each interviewee analyzes at least ten bug reports (O4: Quality bug reports), classifying them into relevant ATD categories, providing a motivation, and following the process outlined in Figure 4 which we described in Section 4.1.

All the sessions were audio recorded for further analysis. We ensured that at least one Architect and one Senior Developer classified each bug report. This allowed us to understand if there were any discrepancies between the viewpoints of architects and developers on architectural issues.

4.2.2 Results and Lessons learned from iteration 2

The recorded sessions were analyzed to understand how the participants reasoned about each bug report classification. This section reports the lessons learnt (O6:

[7] `https://www.nltk.org/`

Lessons learnt) and results (O7: Results from classification from process v2.0) from iteration 2 in Figure 3.

We estimated to spend around five minutes per bug report and to classify at least ten bug reports during the one-hour session per person. However, in reality, it consumed more than five minutes per bug report, and in some cases, took even more than 10 minutes. One participant mentioned that they had not been looking at the bug reports from an ADT point of view before, so it was a new angle to look at the defects. On a few occasions, the participants were uncertain if a defect was related to ATD, since the granularity of architectural components that makes up the system was unclear. 19 bug reports were classified at least by one interviewee. Out of these 19, 16 bug reports were classified by one architect and one senior developer. Out of these 16 bug reports, only six bug reports were classified by two participants into the same ATD categories.

From these preliminary results, it was evident that our idea of identifying ATD from bug reports worked to some extent based on the current classification process. However, the classification was too time-consuming, considering a typical inflow of 150 to 200 bug reports per quarter for System A and the ATD analysis to be conducted quarterly. Though we did not calculate the Return on Investment (ROI) involved in the ATD classification versus the benefits it provides to a software development organization, it was evident from our classification experience that it is challenging to motivate the management to allocate expensive resources such as the software architects and senior developers for extended periods of time. The process also suffered from a lack of reliability and consistency due to the lack of agreement on the scope and boundaries of the software architecture.

We learnt the following lessons (O6: Lessons learnt) from iteration 2, which we used as input to improve our classification process in iteration 3.

Lesson4 The ATD classification process should be efficient enough for industry adoption.
Lesson5 The scope and the boundaries of the architecture components should be clearly defined and agreed upon with the stakeholders.

4.3 Iteration 3

To improve the efficiency of the classification process (Lesson4), we explored the possibility to discard the apparent bug reports that are irrelevant to architecture to filter out only the architecturally significant bug reports. To improve the reliability and consistency of the process and to guide engineers how to systematically identify ATD from bug reports, we introduced the defect types that are described in the Orthogonal Defect Classification (ODC) (Chillarege et al. 1992) into our classification process. To improve the consensus of the software architecture of the system based on Lesson5, we introduced the C4 model[8].

4.3.1 Defect Categories

According to the ODC guidelines, the ODC defect types are defined based on how defects are fixed. For our research, we use the defect types (I3: ODC types relevant for Architecture) defined by Chillarege et al. (1992), listed in Table 4.

[8] https://c4model.com/

Table 4 ODC defect types (Chillarege et al. 1992) and relation to C4 model

ODC defect type	Definition	C4 level
Interface	The defect was the result of a communication problem between subsystems, modules, components, operating system, or device drivers, requiring a change, for example, to macros, call statements, control blocks, parameter lists, or shared memory. Note: A defect that is the result of passing the wrong type of variable is an interface defect, while a defect that is the result of passing the wrong value is an assignment.	2, 3
Function	The defect was the result of the omission or incorrect implementation of significant capability, end-user interfaces, product interfaces, interface with hardware architecture, or global data structure(s).	2, 3
Build/Package/Merge	The defect was encountered during the system build process, and was the result of the library systems or with management of change or version control.	4
Assignment	The defect was the result of a value incorrectly assigned or not assigned at all. Note that a failure correction involving multiple assignment corrections may be of type Algorithm.	4
Documentation	The problem was the result of an error in the written description contained in user guides, installation manuals, prologues, and code comments. Note this should not be confused with an error or omission in the requirements or design that might be a Function or Interface defect type.	not considered as part of the C4 model levels.
Checking	The defect is the result of the omission or incorrect validation of parameters or data in conditional statements. Note a fix involving the correction of multiple checking statements may be of type Algorithm.	4
Algorithm	The defect is the result of efficiency or correctness problems that affect the task and can be fixed by (re)implementing an algorithm or local data structure.	4
Timing/Serialization	The defect is the result of a timing error between systems/subsystems, modules, software/hardware, etc. or is the result of the omission or an incorrect use of serialization for controlling access to a shared resource.	4

We observed in iteration 2 that there was confusion among the interviewees regarding the boundaries and scope of different architecture components, as we reported in Lesson5. We addressed this problem by introducing the C4 model.

4.3.2 Software Architecture and Scope

One of the main challenges with software architecture is to define the architecture and the scope of the architectural components in a given system context. There is no consensus on what software architecture is, and what its boundaries and scope are (Solms 2012). There have been several attempts in the past to define software architecture. Several definitions of software architecture exist, as summarized by the Software Engineering Institute (SEI 2010). The definitions are classified as modern, classic, and bibliographic. Table 5 shows the "Modern" definitions as they are closely associated with the current architecture practices.

Table 5 Modern Software Architecture Definitions

Source	Architecture Definition
Documenting Software Architectures: Views and Beyond (Bass et al. 2010)	"The set of structures needed to reason about the system, which comprises software elements, relations among them, and properties of both." (SEI 2010)
Software Architecture in Practice Bass et al. (2003)	"The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them." (SEI 2010)
ANSI/IEEE Std 1471-2000, Recommended Practice for Architectural Description of Software Intensive Systems 875 (2000)	"Architecture is defined by the recommended practice as the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution." (SEI 2010)

The abstract nature of these definitions leaves room for different interpretations of what architecture means to individuals. Moreover, from a practical point of view, these definitions do not provide adequate guidance to address the main challenge of determining the scope and boundaries of different architectural components that constitute a software system. Hence, it is vital to agree on the criteria to define the boundaries of components that make up the software system with stakeholders and the software architecture team.

In different contexts, the definition and the scope of the architecture vary based on the stakeholders. At Ericsson for example, product owners have a broad definition of architecture, including commercial features and software packaging. To properly navigate the scope of the architecture with different stakeholders, we need to introduce pre-fixes to the term architecture, such as commercial architecture, software architecture, delivery-artifact-architecture, and deployment architecture. Just calling it architecture overloads the term.

On some occasions, at Ericsson we also use the term "logical" architecture with stakeholders such as product owners. In such a case, the architecture refers to a higher-order functional architecture, closer to features or functions, almost like a black-box view of the software system.

To analyze defects for ATD identification, we needed to agree on what software architecture means for the architects and senior developers and use that definition/scope consistently. It is important that all stakeholders are on the same page regarding the scope of the architecture from a ATD classification perspective as well as making decisions based on the identified ATD. Lack of agreement can lead to debates throughout the process without being able to take concrete actions. Existing definitions of software architecture do not anchor at a particular level of system abstraction.

4.3.3 C4 model

Considering the above limitations on the architecture definitions and the industrial needs, we looked at Brown (2011)'s C4 model, where the architecture is described in four levels as outlined in Table 6, facilitating traversal between levels. The level-based approach provides the possibility for different stakeholders to gather the right level

Table 6 Levels of the C4 model

C4 level	Name	Description	Stakeholders
Level 1	System Context	The System context diagrams describe the software system in its surroundings and depicting the external systems and actors that it interacts with.	Anyone that is interested in understanding the system on a high level such as product managers, software architects, new comers to the organization and even external parties.
Level 2	Container	A user has the possibility to zoom into the system to understand the technical building blocks that constitutes the software system and their interactions. A container is a representation of an entity that hosts code or data that usually needs to be running for the system to function. These can be for example, a web application running on Apache Tomcat [a], a java application running in the JAVA Virtual Machine(JVM), and a Node.js [b]application	Product owners, Software Architects, Seniors designers
Level 3	Component	Zooming into a container shows the components that makes up the container. A component can be a grouping of related functionality interactive over well-defined interfaces. For example, in a Java or C# based application, a collection of implementation classes behind an interface can be visualised in a Component diagram.	Software Architects and developers.
Level 4	Code	Code level shows the implementation details describing the classes within a Component and how they interact.	Software Architects and developers.

[a] `http://tomcat.apache.org/`
[b] `https://nodejs.org/en/`

of detail of the software system depending on their role. As observed by Vázquez-Ingelmo et al. (2020), the hierarchical levels in the C4 model support practitioners to understand and ease the comprehension of the software architecture. We decided to use level 2 (container level) and level 3 (Component level) for classifying bug reports (I4: Layers from C4 model relevant for Architecture), as the critical decisions about the system architecture are made on these two levels.

By looking through the lens of the C4 model into the system architecture and the possible defects associated with level 2 and level 3, we identified that ODC defect types "interface" and "function" are closely related to architecture and thereby related to ATD. One could also consider "Algorithm" and "timing/serialization" types associated with the architecture. Our process does not forbid the use of additional defect types, so a practitioner could include different or additional defect types if deemed necessary. The ODC defect types and the C4 levels where such defects could occur are shown in Table 4. Following are some examples of architectural defects on level 2 and 3 that may contribute to ATD.

Containers (level 2):

1. Interface: The defect was the result of a communication problem between containers, due:
   (a) incompatible interfaces design (e.g., REST interfaces exposed by service)
   (b) incompatible database schema/database type
   (c) incompatible or sub-optimal selection of technologies (e.g.: choosing a web container that does not fulfill the desired characteristics such as robustness and performance)
2. Function: The defect was the result of an incorrect or hijacking responsibility or another container.

Components (level 3):

1. Interface: The defect was the result of a communication problem between components within a containers due to:
   (a) incompatible component interfaces design (e.g.: java interfaces)
   (b) incompatible or sub-optimal selection of 2nd / 3rd party libraries within a component.
2. Function: The defect was the result of incorrect or sub-optimal decomposition of responsibilities between components within a container.

4.3.4 MultiDimER v2.0

In iteration 2, we considered bug reports spanning the whole system. Drawing random bug reports from the whole system makes it difficult to derive accurate conclusions from an ATD perspective. To mitigate this problem, we developed the idea of grouping the defects into different architectural components and analyzing the defect clusters belonging to these parts of the system. We implemented tracing commit identities from bug reports and identified the impacted architectural components by extending the MultiDimER tool (A7:Extended python program to automatically classify bug reports into architectural components) that we introduced in iteration 2. This automated classification (O3: MultiDimEr v1.0) improved the process by highlighting the components that resulted in most defect fixes. This strategy provides a means to narrow down and prioritize components that need analysis instead of investigating the whole system.

4.3.5 Results and Lessons learnt from iteration 3

We analyzed 251 bug reports from four different architecture components over four releases of the system that spanned between October 2011 to December 2020. From this list, 53 were categorized as ODC types Interface and Function. We further analyzed these 53 bug reports to determine if they were ATD related and categorized them into one of the ATD categories.

Table 7 summarizes the defect distribution over the four system components and the four releases used in the analysis. From Table 7, we can see that in component "A", 27 out of 64 defects were classified as Interface and Function according to ODC. Seventeen out of 27 defects were further classified and categorized to ATD categories.

Additionally, we report in Table 8 the time spent on different phases of the classification process. The efficiency of the process is a crucial factor for the process to be adopted in the industry.

Table 7 Defects distribution over Architecture components

Release	Component A	Component B	Component C	Component D	Total
Release 1	4	6	3	10	23
Release 2	10	20	17	30	77
Release 3	30	15	20	0	65
Release 4	20	20	30	16	86
Total	64	61	70	56	251
Interface and Function (I&F)	27	7	10	9	53
ATD defects	17	5	2	6	30
(ATD defects)/(I&F) proportion	63%	71%	20%	67%	57%
(ATD defects)/Total proportion	27%	8%	3%	11%	12%

Table 8 Average time spent on process phases

Phase	Average time required (minutes)
Defect mapping to architectural components	0 (Automated by the MultiDimEr tool)
Mapping to ODC Category and C4 model	3
Mapping to ATD Category	3

RQ4 What are the lessons learned from designing an ATD identification process from bug reports?

- Lesson1: ATD categories should be anchored in the context of the software system under consideration.
- Lesson2: ATD categories should not significantly overlap with one another.
- Lesson3: Bug reports should contain rich information in terms of describing the issue, problem analysis, and the implemented solution to be usable for ATD identification.
- Lesson4: Overall, ATD classification process should be efficient enough for industry adaptations.
- Lesson5: The scope and the boundaries of the architecture components should be clearly defined and agreed upon with the stakeholders.

4.4 The ATD classification process

We summarize here the resulting ATD classification process and the key concepts that were refined after three design iterations. Our process consists of the following concepts and the specific implementation of our process within the current study:

1. **CT1. Consensus of software architecture abstraction levels and their boundaries:** As we pointed out in Section 4.3.2, there is a need to clearly define and agree on different architecture components and their boundaries among the different stakeholders. In our implementation of the process, we utilized the C4

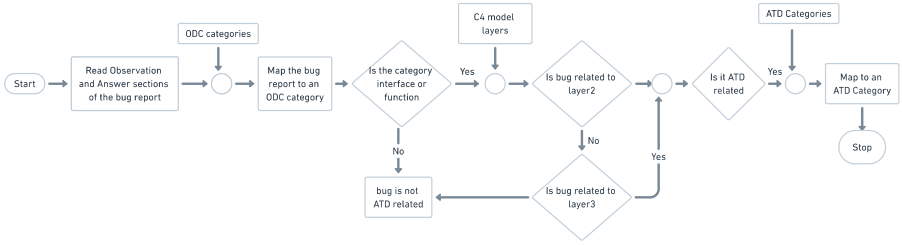

Fig. 6 ATD identification process

model as a way of communicating and defining the architecture abstractions and boundaries.

2. **C2. A well-defined set of defect categories:** Grouping defects into categories helps to filter out defects that are related to architecture. We utilized the defect categories from ODC (Chillarege et al. 1992). The defects defined in the ODC are to a large degree non overlapping, and we used them without introducing additional categories.
3. **CT3. A well-defined set of ATD categories:** ATD categories are used as a method of grouping ATD-related bugs. We conducted a literature review to identify these categories from prior research, which are listed in Figure 5 and described in Table 3.
4. **CT4. A mechanism to group defects to architecture components:** We automated the grouping of bug reports to architecture components through a python based tool (Silva et al. 2022).

.

The overall ATD classification process and how the above building blocks are being used are depicted in Figure 6. The process starts by reading the observation and the answer sections of the bug reports, similar to the process outlined in Section 4.1. The difference compared to the process in iteration 1 and iteration 2 is that now there are two intermediary classifications that need to be performed, to systematically guide towards an ATD classification. The first intermediary classification is to map the bug report to an ODC defect category from Table 4. Then, in the second intermediary classification, the bug report is mapped to a C4 layer from Table 6. After these classifications, the final ATD classification can be performed by reasoning around the ODC defect type and the C4 layer.

Concluding iteration 3, we implemented the process with an industrial data set that has led to insights about System A.

5 Static Validation

As suggested by Wieringa (2014), the designed process was put into test in an empirical context towards the end of iteration 3. The final version of the process was implemented at Ericsson with the involvement of the lead author and participants P1, P6 and P7. This step corresponds to the static validation step in the technology transfer model by Gorschek et al. (2006).

We created a classification guideline template based on the steps outlined in Figure 6. The template consisted of the steps to be followed to classify a bug report, the list of ODC defect types and their descriptions, as shown in Table 4, a list of the C4 model layers and their descriptions, and the list of bug reports to be classified. The template was reviewed by the three authors before its application. The final version of the template was also reviewed by participant P1.

The template was given to participants P6 and P7, who independently classified 27 bug reports from System A. The goal was to test the accuracy and efficiency of the process without any substantial training efforts. Participants P6 and P7 were not involved in other iterations during process development. Furthermore, they were also the most junior participants with the least working experience at Ericsson. The motivation for selecting P6 and P7 is to evaluate if we can delegate the intermediary classification stages to developers without having to rely on senior developers and architects who are usually occupied in critical design activities. The classification results showed that both participants P6 and P7 classified 16 bug reports to the same ODC category and to the same C4 levels. The results were validated by the lead author.

5.1 Architectural Components

In this section, we provide an overview of the four components of System A. The component names have been anonymized due to a Non Disclosure Agreements (NDA) with the company. This overview provides the context to the ATD defects analysis in Section 5.2.

Component A: Northbound Interface (NBI) service: This component exposes several interfaces based on Hypertext Transfer Protocol (HTTP) and Telnet protocols and is the entry point to the system. The component is based on Apache Karaf[9] which is a polymorphic application container powered by Open Services Gateway initiative (OSGi)[10] framework. OSGi is a specification that describes a modular framework for the JAVA language. This component also provides a Software Development Kit (SDK) for developing new customer specific NBI interfaces by system integrators.

Component B: Logging: The logging component is responsible for writing transaction logs from different components within the system. It accepts requests/responses that should be written to a shared NoSQL database during the traffic flow. It also provides an API for retrieving transaction logs, for example, to the GUI component that is used for troubleshooting.

Component C: Platform: The platform component was introduced to classify issues related to the deployment, upgrades, and operating system.

Component D: Core Engine: This is the largest and most important component of the system where the business logic resides and is the oldest component in the system which has evolved for more than ten years. Some of the functionality has been extracted into separate services over time. However, there is still work to be done in future architecture evolution to continue to modularize this core engine.

[9] `https://karaf.apache.org/`
[10] `https://www.osgi.org/`

In the next section, we describe the identified collective symptoms present in the bug reports, the causes of defects from an architectural perspective, and discuss the consequences of ATD defects.

5.2 Identified ATD categories and defects patterns

By applying the classification process, we identified 30 ATD-related defects. We categorized them into five ATD categories, based on their commonalities.

What follows next are the details of five ATD categories that resulted from classifying the 30 ATD-related defects. We structure each ATD category by (a) providing the symptoms according to bug reports, (b) results of an analysis by a product expert based on the developers' descriptive text with regard to the cause of the issue and how it was fixed, followed by (c) the analysis of the consequences of the defects on the product. The defects were mapped to an ATD category based on the causes of the defects from an architecture perspective. Based on the causes, we looked up to find a matching ATD category from Table 3.

ATD category - 1.1 module dependencies: We defined module dependencies as the behavior of one module or component that influences the behavior of another component in an undue manner.

Symptoms from the bug reports: When shutting down Components A and B, errors were visible in the components' logs, indicating that they failed to check the leader election state due to not being able to connect to the NoSQL database. There was no traffic impact due to this issue, but it was confusing for the users to understand the impact of the errors being logged and their impact on the overall system.

Causes from an architecture point of view: When shutting down several modules at the same time, a process called the "leader election" was triggered. Leader election is a mechanism used in distributed computing to designate a single process to coordinate some task distributed among several computers (nodes). During the leader election process, components A and B tried to connect to a NoSQL database that is being shared among multiple components, but the database itself is shutting down, leading to a connection to failure.

Consequences due to the defect: As a workaround solution, the startup/shutdown of the NoSQL database was synchronized with the election process. This avoids that the startup/shutdown and leader election occurring at the same time. However, this workaround could potentially create a deadlock situation, which is worse than seeing logs in errors.

The coupling between critical processes is problematic from an architectural point of view. This issue still needs to be addressed in a future release. A deeper analysis is required to decompose the system to decouple leader election processes from startup/shutdown sequences.

ATD category - 1.2.1 - unsound dependencies towards frameworks: We defined unsound dependencies towards frameworks when the chosen frameworks are not fit for the purpose, or unsound from an architectural design perspective.

Symptoms from the bug report:

Component A allows the deployment of customer-specific NBI bundles. In older versions of the system, the bundle is activated if it gets deployed successfully. In newer versions of the system, the deployed bundle is merely marked as "installed" and not activated automatically. This made the service exposed by the bundle unavailable, resulting in traffic failures.

Causes from an architecture point of view: In newer versions of Component A, an internal logging library included a dependency towards javax.servlet-api-3.1.0. The customer's new interface bundle deployed into Component A also included a dependency on the internal logging library. However, the OSGi import-packages included javax.servlet and javax.servlet.http version 4, which is incompatible with 3.1. Since the new module does not import version 4 of javax.servlet in karaf, the bundle was inactive.

Component A, where the fault was fixed deviates from the design pattern used in the rest of the system concerning web frameworks. The architects decided not to use web frameworks in this component since they provide little value to the use cases at hand. It was known from an early stage that OSGi is problematic, and it is not fully compliant with the design patterns being defined for the product. In this particular issue, it was library conflicts between our own bundles and what was provided by the Karaf framework.

Consequences due to the defect: To work around this restriction, javax.servlet old version had to be excluded from the bundle. Karaf framework was removed in a later release of the system during system modernization.

ATD category - 3.3 - Violations of standards from standardization bodies Defined as a sub-category within the Compliance Violations, the violation of standards refers to compliance issues with regard to standards specifications defined by standardization bodies such as the IETF, W3C and 3GPP, for example.

Symptoms from the bug reports: The system contains a GUI to visualize the transaction logs (inbound requests/responses and southbound requests/responses with different protocols). This functionality is mainly used for troubleshooting. For certain types of business logic transactions, it was not possible to retrieve the southbound logs via the GUI. When tried to fetch logs, the responses were empty for certain protocols. The users had to get data directly from the database using a command-line interface (CLI) tool and analyze the logs manually due to this bug.

Causes from an architecture point of view: This issue is related to Component B. The writing and reading of transaction logs were not correctly implemented. The log was stored and processed in Comma-separated values (CSV) format. If the request or response contained special characters such as a comma, double quotes, and similar the code tried to escape these characters. However, the implementation was not complete and did not cover all special cases.

The system had its own implementation of the Comma-separated values (CSV) and was not compliant with a particular CSV standard (for example, RFC 4180[11]).

Consequences due to the defect The fix was to use an open-source library to replace the own implementation. It might have been possible to detect the need for a CSV parser and select an appropriate library during the architectural impact analysis. Since then, to minimize this type of issue, the "implementation proposal template" (Fricker et al. 2010) has been updated to include a dedicated section to

[11] `https://datatracker.ietf.org/doc/html/rfc4180`

identify the need for external libraries if the implementation involves well-known standards/protocols. The implementation proposal is a document that the development teams are expected to produce, and get approval from the Architecture team. The goal of this document is to handshake the understanding of the requirement and the high-level solution proposal outlined by the product owners by the development team and also to agree on the software design details.

ATD category - 3.4 - Violations of security principles: Another sub-category within Compliance Violations is violations of security principles; defined as not following security best practices such as the OWASP top ten and CWE top 25.

Symptoms from bug reports: In Component A, the SOAP API was vulnerable to XML Attribute Count Attacks. This issue can be caused when the XML parser improperly parses all XML content before validating it, increasing the risk of denial of service due to excessive system resource usage.

Another defect was that Component A supports TRACE and/or TRACK methods. TRACE and TRACK are HTTP methods that are used to debug web server connections and should have been disabled.

Both defects led to an increased probability that an attacker could penetrate the system.

Causes from an architecture point of view: The code base related to the XML issue is over ten years old. The system evolved over the recent years to be deployed in cloud environments, instead of in dedicated hardware. This trend has significantly increased security testing both by the design organization as well as the customers that now discover issues with old code bases.

Exposure of TRACE/TRACK methods was present in a relatively new development. However, the security aspects lacked a thorough analysis.

The security requirements are under-specified and are often non-compliant with security standards as the system evolves. There is a lack of a Security Architect that oversees the overall system security, including legacy code.

Consequences due to the defect: To fix the XML issue, a configurable request size was introduced which was set to a high enough value for the request sizes that the development organization was aware of. Though this reduces the attack surface by minimizing the number of attributes to be parsed, it is not an optimal solution. A proper solution is to add thorough interface validations before the XML parser parses the XML content.

To address the issue, the code was updated to block TRACE/TRACK methods.

ATD category - 3 - non-compliant interfaces: We define a non-compliant interface as an internal or external interface that deviates from its specification in a manner that breaks the communication between entities interacting through the interfaces.

Symptoms from the bug reports: When sending a request with an alphanumeric transaction identifier through Component A, the logging component (Component B) did not store the id. However, if a numeric transaction identifier was sent, then the id was stored in the database. In the interface schema, the transaction identifier was of type xs:string, hence the logging component should have captured it.

TransactionId is used to correlate requests and responses that are part of a single transaction. The lack of the transactionId makes it difficult to troubleshoot the system.

Causes from an architecture point of view: The transaction identifier was handled as an Integer in Component A, the NBI. When a non-integer value was received, it could not translate it to an integer, and hence ignored the identifier completely. The code base that required updates was older than ten years. It could be that the intention from the beginning was to use only numeric characters for the transaction identifiers, but as the solutions evolved over the years, the external systems interfacing with our system may have started using alphanumeric identifiers as well.

Consequences due to the defect: Changes were required in Components A and D to properly support the alphanumeric characters in the transaction identifier. During system evolution and major architecture changes, the main focus is to get the feature set working on par with the old system, overseeing details on existing interfaces. The implementation proposal template has been updated recently with a dedicated interface section that could detect interface incompatibilities in the early design phases. The template was revised during decomposing of System A into microservices, since backward compatibility of interfaces is a cornerstone for the reusability of microservices in different contexts and for smooth upgrades/rollbacks.

6 Discussion

The classification results show that the identified 30 ATD-related defects were mapped to five out of 16 ATD categories from Table 3. The mapping to only five categories could be due to the fact that the bug report sample that we analyzed did not contain any bug reports pointing to other ATD categories. It may not necessarily mean that other ATD categories do not exist in System A.

On some occasions, we observed that certain defect fixes resulted in updating several architectural components. We suspected this might be due to unsound dependencies between components. However, the analysis revealed that defects that required updating multiple architecture components were caused by changes in the shared libraries version where the defects were resolved.

There are several advantages of using our approach to identify potential ATD. Clustering the bug reports into different architecture components helps to identify weaker components in terms of defect generation that can be prioritized for ATD analysis. This step makes the process more efficient and economical by narrowing down the components that require deeper analysis than investing in full-scale system analysis.

Another advantage of our process is that no extensive upfront preparation is required. Since defect analysis is usually a part of the software development processes, our process can easily be integrated into existing defect analysis processes (e.g. FST analysis (Damm et al. 2006)) and provide an architectural perspective. As we showed in our classification results in Iteration 3, the process's efficiency makes it more likely to be adopted in the industry.

Our classification process consists of four building blocks: ATD categories, defect categories, an architecture representation method, and a tool to map bug reports to architectural components. When adopting our process, users are encouraged to adjust it according to their needs by using other mechanisms than the ones we used in this study. ATD categories such as insufficient metadata in the messages, microservice coupling, and lack of communication standards, as observed by de Toledo et al.

(2021) can be used for ATD detection in microservices. In the defect classification stage, a different technique than ODC to classify bug reports could be used, and the C4 model can be replaced with another model to reason about the architecture.

A commonality that can be observed in our approach and some of the other non-source code-based ATD identification approaches (de Toledo et al. 2021; Li et al. 2015) is the need for expert knowledge of the system under investigation. It can be argued that the need for system expertise is a limitation. However, it is not easy to replace that knowledge with other means in the software architecture context. This expert knowledge usually exists within an architecture team and senior designers. The management ensures the continuation of such expertise within the organization for future system development. Hence, we believe that the system architecture knowledge within a software organization that is required by our ATD identification process is available in most organizations. Moreover, the architecture expertise is only needed once the bulk of the defects has been filtered out by the early phases of our process using ODC and C4 levels.

Architecture Decision Records (ADRs) are valuable assets, as suggested by Li et al. (2015), that utilized ADRs and change scenarios together with interviews to detect ATD. However, this approach requires upfront preparation and creating a list of architecture decisions and change scenarios according to a template. Combining ADRs with our approach can be an added improvement for ATD detection. Organizations that enforce ADRs can utilize them to extend our process by comparing the relevant ADRs against the ATD found by our process. The ADR knowledge base can also be used to delegate the ATD analysis to other roles than the architects and senior developers and make the process less reliant on scarce resources such as software architects and senior developers.

Results from the final process implementation at Ericsson, as shown in Table 8 indicate that, on average, we managed to classify a bug report to an ODC Category and a level in the C4 model within three minutes. This was a significant improvement compared to the classification of around eight bug reports per hour in iteration two. The strategy to eliminate the obvious bug reports that are irrelevant from an ATD perspective was the main reason for this efficiency gain.

The main issues we encountered in the classification are the definition and the scope of the software architecture of a given system. Even though a few definitions of software architecture exist, they are abstract and subjective, and can hardly be used for practical, classification purposes. The ATD concept also inherits and suffers from the lack of a clear definition of architecture. The scope or boundaries of the architectural components are more critical for our classification than a debate over software architecture. We mitigate this problem by introducing the C4 model. For the process to be practical, the architecture stakeholders must agree on the boundaries of different architectural components that constitute a software system.

We defined ATD categories to classify the defects. It is worth noting that in some studies (Verdecchia et al. 2021, 2018), these categories are referred to as ATD items, whereas in other studies, such as the unified model of ATD proposed by Besker et al. (2018), these are referred to as ATD categories. The differences between these terminologies however do not impact our process.

6.1 Limitations

We identified that high-quality bug reports are crucial for the efficient process application. To filter out descriptive bug reports, we counted the number of sentences as a simple measure. It may be possible to introduce more sophisticated mechanisms that analyze the content of bug reports in more detail to filter out low-quality instances; we leave such an investigation for future work.

The writing style and the way individuals express their views vary. Ericsson is a large software development organization where software development is spread over multiple countries. The cultural background also influences language expressions, which can be observed in the bug reports. We inspected the defect-fixing patch sets to understand the solutions better when encountering ambiguity issues due to language expressions.

We restricted ourselves to the four main ATD categories in this study, but the classification process itself is not dependent on the number of ATD categories. However, it is possible to define more categories depending on different study contexts, provided that the ATD categories do not significantly overlap with one another.

Architects/Designers can introduce ATD due to biases such as confirmation bias, anchoring bias, and the curse of knowledge bias, as pointed out by Borowa et al. (2021). ATD analysis can suffer from such biases since the experts that made the architectural decisions may now be involved in the postmortem. Biasing is not specific to our study but rather a common problem when a human is involved in the investigation. We minimized potential confirmation bias by using two persons to classify a single bug report.

7 Conclusions

Our goal in this study was to design an ATD identification process using bug reports as the primary data source. To the best of our knowledge, there is no prior research that explicitly utilizes bug reports as a data source for ATD identification. We designed the ATD identification process, tested and refined it in three design iterations. The findings from our research indicate that it is possible to use bug reports to identify ATD.

We used design science to iteratively refine the process. From our experience of implementing the classification process at Ericsson, we believe that this process could be scaled in an industrial context, contributing to bootstrapping an ATD management process in practice. Based on the lessons learned from the design iterations, we emphasize the need for agreement among the stakeholders regarding the scope and boundaries of architectural components. Visualizing the software architecture through the lens of the C4 model provided us with a practical way to reason about software architecture on different abstraction levels. In our study, we focused on ATD associated with level 2 and level 3 according to the C4 model, where most of the architecture design decisions occur. The ODC helped us efficiently filter out the essential bug reports that may give indications of ATD.

We recommend using C4 levels to visualize and define the scopes of software architectures in the ATD study contexts. The layering approach helps formalize the required architecture abstraction levels in different ATD study contexts, which helps compare different ATD studies. We believe this is better than the current

trend of defining ATD differently in different studies. Exploration of communicating software architecture with the right level of abstraction to different stakeholders is an important research direction based on our findings.

To improve process efficiency, we plan to explore the possibilities to automate the ODC classification utilizing machine learning techniques.

We hope that our ATD identification process contributes to a novel and more pragmatic approach to identify ATD, complementing the existing methods, and opening new research directions to improve ATD related studies and make ATD identification more relevant to the industry.

Data Availability Statement

Research data (trouble reports and their analysis) are not shared as they contain proprietary/sensitive information.

Acknowledgments

We would like to thank Ericsson AB and the KKS foundation for supported this research through the S.E.R.T. Research Profile project at Blekinge Institute of Technology. We would also like to thank the architects, developers and the management at Ericsson for their participation in the interviews and feedback.

Conflict of interest

The authors declare that they have no conflict of interest.

References

(2000) Ieee recommended practice for architectural description for software-intensive systems. IEEE Std 1471-2000 pp 1–30, DOI 10.1109/IEEESTD.2000.91944

Bass L, Clements P, Kazman R (2003) Software architecture in practice. Addison-Wesley Professional

Bass L, Merson P, Bachmann F, Stafford J, Clements P, Nord R, Garlan D, Ivers J, Little R (2010) Documenting software architectures: views and beyond. Addison-Wesley Professional

BenIdris M, Ammar H, Dzielski D, Benamer WH (2020) Prioritizing software components risk: Towards a machine learning-based approach. In: Proceedings of the 6th International Conference on Engineering & MIS 2020, pp 1–11

Besker T, Martini A, Bosch J (2018) Managing architectural technical debt: A unified model and systematic literature review. Journal of Systems and Software 135:1–16

Borowa K, Zalewski A, Kijas S (2021) The influence of cognitive biases on architectural technical debt. In: 2021 IEEE 18th International Conference on Software Architecture (ICSA), IEEE, pp 115–125

Brown S (2011) The c4 model for visualising software architecture. URL: https://c4model com/(visited on 08/27/2020)

Chang CP, Chu CP, Yeh YF (2009) Integrating in-process software defect prediction with association mining to discover defect pattern. Information and software technology 51(2):375–384

Chillarege R, Bhandari IS, Chaar JK, Halliday MJ, Moebus DS, Ray BK, Wong MY (1992) Orthogonal defect classification-a concept for in-process measurements. IEEE Transactions on software Engineering 18(11):943–956

Cunningham W (1992) The wycash portfolio management system. ACM SIGPLAN OOPS Messenger 4(2):29–30

Damm LO, Lundberg L, Wohlin C (2006) Faults-slip-through—a concept for measuring the efficiency of the test process. Software Process: Improvement and Practice 11(1):47–59

Dehaghani SMH, Hajrahimi N (2013) Which factors affect software projects maintenance cost more? Acta Informatica Medica 21(1):63

Ernst NA, Bellomo S, Ozkaya I, Nord RL, Gorton I (2015) Measure it? manage it? ignore it? software practitioners and technical debt. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, pp 50–60

Fontana FA, Pigazzini I, Roveda R, Tamburri D, Zanoni M, Di Nitto E (2017) Arcan: A tool for architectural smells detection. In: 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), IEEE, pp 282–285

Fricker S, Gorschek T, Byman C, Schmidle A (2010) Handshaking with implementation proposals: Negotiating requirements understanding. IEEE software 27(2):72–80

Garcia J, Popescu D, Edwards G, Medvidovic N (2009) Identifying architectural bad smells. In: 2009 13th European Conference on Software Maintenance and Reengineering, IEEE, pp 255–258

Gorschek T, Garre P, Larsson S, Wohlin C (2006) A model for technology transfer in practice. IEEE Software 23(6):88–95, DOI 10.1109/MS.2006.147

Kruchten P, Nord RL, Ozkaya I (2012) Technical debt: From metaphor to theory and practice. Ieee software 29(6):18–21

Li Z, Liang P, Avgeriou P (2015) Architectural technical debt identification based on architecture decisions and change scenarios. In: 2015 12th Working IEEE/IFIP Conference on Software Architecture, IEEE, pp 65–74

Martini A, Fontana FA, Biaggi A, Roveda R (2018a) Identifying and prioritizing architectural debt through architectural smells: a case study in a large software company. In: European Conference on Software Architecture, Springer, pp 320–335

Martini A, Sikander E, Madlani N (2018b) A semi-automated framework for the identification and estimation of architectural technical debt: A comparative case-study on the modularization of a software component. Information and Software Technology 93:264–279

Mumtaz H, Singh P, Blincoe K (2020) A systematic mapping study on architectural smells detection. Journal of Systems and Software p 110885

Parnas DL (1972) On the criteria to be used in decomposing systems into modules. In: Pioneers and their contributions to software engineering, Springer, pp 479–498

Patton MQ (2001) Evaluation, knowledge management, best practices, and high quality lessons learned. American Journal of Evaluation 22(3):329–336

Pérez B, Correal D, Astudillo H (2019) A proposed model-driven approach to manage architectural technical debt life cycle. In: 2019 IEEE/ACM International Conference on Technical Debt (TechDebt), IEEE, pp 73–77

Runeson P, Höst M (2009) Guidelines for conducting and reporting case study research in software engineering. Empirical software engineering 14(2):131–164

SEI (2010) What is your definition of software architecture URL `On-lineat:https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=513807`, accessed: 2021-10-10

Silva L, Unterkalmsteiner M, Wnuk K (2018) Monitoring and maintenance of telecommunication systems: Challenges and research perspectives. In: KKIO Software Engineering Conference, Springer, pp 166–172

Silva L, Unterkalmsteiner M, Wnuk K (2022) Multidimer: A multi-dimensional bug analyzer. In: 2022 IEEE/ACM International Conference on Technical Debt (TechDebt), IEEE Computer Society, pp 66–70

Solms F (2012) What is software architecture? In: Proceedings of the south african institute for computer scientists and information technologists conference, pp 363–373

de Toledo SS, Martini A, Sjøberg DI (2021) Identifying architectural technical debt, principal, and interest in microservices: A multiple-case study. Journal of Systems and Software 177:110968

Vázquez-Ingelmo A, García-Holgado A, García-Peñalvo FJ (2020) C4 model in a software engineering subject to ease the comprehension of uml and the software. In: 2020 IEEE

Global Engineering Education Conference (EDUCON), IEEE, pp 919–924
Verdecchia R, Malavolta I, Lago P (2018) Architectural technical debt identification: The research landscape. In: 2018 IEEE/ACM International Conference on Technical Debt (TechDebt), IEEE, pp 11–20
Verdecchia R, Kruchten P, Lago P, Malavolta I (2021) Building and evaluating a theory of architectural technical debt in software-intensive systems. Journal of Systems and Software 176:110925
Wieringa RJ (2014) Design science methodology for information systems and software engineering. Springer
Wu LL, Xie B, Kaiser GE, Passonneau R (2011) Bugminer: Software reliability analysis via data mining of bug reports
Xiao L, Cai Y, Kazman R, Mo R, Feng Q (2016) Identifying and quantifying architectural debt. In: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), IEEE, pp 488–498
Xiao L, Cai Y, Kazman R, Mo R, Feng Q (2021) Detecting the locations and predicting the costs of compound architectural debts. IEEE Transactions on Software Engineering