Identifying and Prioritizing Architectural Debt through Architectural Smells: a Case Study in a Large Software Company

Antonio Martini^{2,3}, Francesca Arcelli Fontana¹, Andrea Biaggi¹,
and Riccardo Roveda⁴

¹ Università degli Studi di Milano - Bicocca, Milan, Italy
`arcelli@disco.unimib.it`,
`a.biaggi@campus.unimib.it`

² CA Technologies, Barcelona, Spain

³ University of Oslo, Oslo, Norway
`antonio.martini@ifi.uio.no`

⁴ Alten Italia, Milano, Italy
`riccardo.roveda@alten.it`

Abstract. Architectural technical debt can have a huge impact on software maintainability and evolution. Hence, different architectural violations, detected as architectural smells, need to be identified and refactored. In this paper, we conducted a multiple case-study on several architectural smells detected in four industrial projects. We conducted an in-depth investigation with a questionnaire, interviews and thorough inspection of the code with the practitioners. We evaluated the negative impact of the technical debt detected by the architectural smells, their difficulty to be refactored and the usefulness of the detection tool. The results show that practitioners appreciated the help of automatic detection, and that they prioritize refactoring architectural debt that causes more negative impact despite the higher refactoring effort.

1 Introduction

Architectural Technical Debt (ATD) can have a huge negative impact on software maintainability and evolution. In a recent survey [1], more than 250 practitioners report that the average time wasted because of the presence of Technical Debt accounts for 37% of the whole development time. From the same study, it is also clear that ATD generates the most negative impact.

ATD is regarded as suboptimal solutions in the architecture of a product. A large number of software components that are too interdependent can be considered as an example of ATD. Suboptimal solutions can cause a negative impact, in the form of extra effort, when maintaining or evolving the project. For example, a component that has a lot of dependencies to many other components, would have ripple effects when changed: every time a bug is fixed or a new functionality is added, the practitioners need extra effort. When this, or other sorts of negative impact, occur, they represent the *interest* of the ATD.

Some ATD can be automatically detected, thanks to identification tools analyzing source code [2,3,4]. More in particular, existing tools recognize Architectural Smells (AS)[5,6], or else anti-patterns present in the architecture of the project. These patterns can be identified, for instance, by analyzing the dependency graph of the project. An AS is usually a *symptom* of the presence of ATD, and detecting AS can help developers and architects identifying ATD.

Finally, there is a cost associated with the removal of ATD, which is regarded as the *principal* of Technical Debt. ATD is removed by paying the principal in term of time for refactoring the code. The main reason to repay the principal, and to refactor ATD, is to avoid paying its interest [7].

In summary, it is critical to detect ATD, but also to prioritize it by understanding its interest and principal. Although some tools are able to detect AS (and therefore pointing at the possible presence of ATD), there are no studies on what negative impact and cost of refactoring are associated to such ATD and how practitioners prioritize ATD revealed through AS.

Given the previous motivations, in this study we aim at answering the following RQs:

- RQ 1: How do AS help practitioners in identifying ATD?
- RQ 2: How do practitioners prioritize ATD revealed through AS?
- RQ 2.1: How is the AS's negative impact (interest) perceived by the practitioners?
- RQ 2.2: What is the refactoring cost (principal) of AS perceived by the practitioners?

By answering RQ1, we aim at understanding if AS are useful to automatically identify ATD in industrial projects. Answering RQ2 means answering the combination of RQ2.1 and RQ2.2: understanding what negative impact is generated by the ATD and what cost of refactoring is required. This would help identifying which AS are more critical for the practitioners to prioritize.

In summary, with this case-study, we make a first step towards understanding how practitioners can semi-automatically detect and prioritize ATD.

The paper is organized through the following sections: in Section 2, we present related works; in Section 3, we describe the design of the case study, while in Section 4, we present the results; in Section 5, we discuss the results with respect to the RQs; in Section 6, we outline the threats to validity and finally in Section 7 we draw our conclusions.

2 Related Work

To the best of our knowledge, no studies have been conducted on identifying and prioritizing ATD using architectural smells. A recent paper [8] reports a comparative case-study on a component modularization, analyzing the negative impact saved by reworking a single instance of Architectural Debt. The study also reports an ad-hoc measurement project to detect the ATD, but it does not assess AS and does not compare several projects.

Several studies on *code smells* have been conducted in collaboration with practitioners, but not on *architectural smells*. An exploratory survey on code smells has been performed by Yamashita et al. [9]: their results showed that a large proportion of developers did not know about code smells. Soh et al. [10] conducted a study where professionals were hired to perform maintenance tasks on functionally equivalent Java projects in order to assess whether code smells affect maintenance activities. Another empirical study on understanding maintenance problems related to code smells, has been done by Yamashita [11]. Palomba et al. [12] conducted a study on developer’s perception of the nature and severity of code smells.

3 Research Design

We conducted a multiple case-study in a large, international company located in Sweden. In particular, we automatically analyzed four projects and we assessed the output with the practitioners responsible for the development and maintenance of such projects.

3.1 Case-study Design

We designed an embedded, multiple case-study, according to the guidelines in [13]. Our unit of analysis was a project developed by the organization. We conducted an in-depth analysis of four software projects of the same company, described in Section 3.2. The investigation included a mix of quantitative and qualitative methods, which is highly recommended in case-studies [13], where it is crucial to reach a detailed understanding of complex systems, such as large software projects.

The objectives of the case-study were:

1. automatically identify the AS in the industrial projects (RQ1)
2. evaluate the output in terms of negative impact (RQ2.1) and cost of refactoring (RQ2.2), to prioritize the ATD.

To identify the AS, we used Arcan [5], a tool for architectural smell detection described in Section 3.3. We decided to use Arcan for: reproducibility purposes since the tool is fully available online, it has been previously validated and the formulas used to identify the smells are well known. This allows other researchers to reproduce this study in other organizations and compare the results with ours.

To answer RQ1 and RQ2, for each project, we performed the following steps:

- We conducted a meeting with the team to understand the project.
- The code was analyzed using Arcan. A sample of the detected AS was then selected to be further assessed. We selected 22 AS since understanding in-depth architectural issues, their negative impact and their cost of refactoring was quite time consuming for the practitioners. This did not allow us to evaluate a larger number of AS. However, 22 cases is a reasonable number

considering the kind of in-depth analysis that was performed. In particular, we extracted the AS of three different kinds in a balanced number for each project. We chose the ones that were considered the most critical, according to Arcan’s severity measure (see Section 3.3). This was done to focus on AS that would be more probably regarded as problematic by the practitioners.

- We conducted a group interview with the team responsible for the analyzed project. We selected participants who had high experience and a higher knowledge of the project. The interview lasted, on average, two to three hours, and included the following activities:
 1. First, we introduced Arcan and the AS to the practitioners. We carefully presented the AS types and we explained how they were calculated (including formulas) and what they were expected to reveal.
 2. Then, we showed a graphical representation of the smell in the form of a graph, rendered with Neo4J⁵, which visualized the involved classes, packages and dependencies.
 3. We asked the developers to identify the architectural issue related to the selected AS. In order to avoid speculations, we asked the practitioners to navigate the source code and to share the screen with the researchers and with the other interviewees.
 4. Once the architectural issue related to the AS was identified, we conducted an assessment according to the objectives defined in our RQs. In particular, we used the questionnaire described in Section 3.4. Such questionnaire was divided in four parts: 1) understanding if the considered smell was pointing at a critical underlying ATD and if it was known 2) assessing the negative impact experienced and estimated by the practitioners. In order to do so, we used a method [14], empirically evaluated, to quantify the negative impact of Technical Debt; 3) assessing the costs and steps necessary for refactoring the AS; 4) understanding what was the current process to evaluate quality and what additional value would the automatic tool give the practitioners.

3.2 Analyzed Projects

The analyzed projects were developed by the same company and they were written in Java: they had different size but they operated in the same field of Product Data Management. A brief description of the projects is outlined in Table 1. The interviewees were nine, two for projects B, C and D and three for project A.

3.3 Data Collection - automatic detection of AS with Arcan

Arcan [5] is a tool for architectural smell detection in Java projects. An architectural smell can derive from commonly used architectural decisions, intentional

⁵ <https://neo4j.com/>

Table 1: Analyzed Industrial Projects and Architectural Smells

Name	Description	Metrics		Architectural Smells					
		NoP	NoC	UD	HL	Class CD	Package CD	AS	
A	Product Data Management (PDM)	269	10171	476	1	199	31	707	
B	After market	240	7261	98	0	7	6	111	
C	Audit project	220	3250	34	1	31	8	74	
D	Warehouse management project	166	3067	53	0	49	7	109	

Legenda: NoP: Number of Packages, NoC: Number of Classes, Unstable Dependency (UD), Hub-Like Dependency (HL), Cyclic Dependency (CD), Total Architectural smells (AS)

or not, that negatively impact internal software quality [15] with large effects on software maintainability [16].

In this study, we considered the following three architectural smells adequately validated in [5,17]:

- **Unstable Dependency (UD)**: describes a subsystem (component) that depends on other subsystem that are less stable than itself, with a possible ripple effect of changes in the project.
- **Hub-Like Dependency (HL)**: occurs when an abstraction has (outgoing and ingoing) dependencies with a large number of other abstractions.
- **Cyclic Dependency (CD)**: refers to a subsystem (component) that is involved in a chain of relations that break the desirable acyclic nature of a subsystems dependency structure. The subsystems involved in a dependency cycle can be hardly released, maintained or reused in isolation. Arcan detects this smell according to different shapes [18].

Moreover, Arcan estimates the *Severity* of each architectural smell [19] according to the values of the metrics used for the AS detection. A *SeverityScore* (AS_k) is evaluated for each instance of AS and according to each type of AS in order to evaluate the criticality of the smell. In our study with the practitioners, we selected the AS instances with the highest Severity, according to each type of AS.

In this work we have considered the above three AS since these AS are detected by an available tool such as Arcan and they are based on dependency issues which certainly represent relevant sources of possible architectural debt. Since this kind of smells do not represent the only source of architectural debt, we will consider in the future other categories of architectural smells. We focused our attention on architectural smells and not on code smells [20], also if some code smells such as for example the God Class smell [21] can have an impact at architectural level and hence it can be considered an architectural smell.

3.4 Data collection - survey with practitioners

The questionnaire carried out during the interviews is reported in Table⁶ in [22]. We used the survey tool Google Forms. We asked the practitioners to fill in a form (the same) for each AS.

Identifying Architectural Debt via Architectural Smells (RQ1) We started asking if the analyzed AS was related to a critical architectural issue (ATD) (Q1.1). Only the ones considered as high priority issues were then further analyzed. This was done because we wanted to obtain more in-depth analysis rather than more sub-cases of smells, to better understand the details of the ATD. Then, we asked if the practitioners were already aware of the AS before it was revealed during the interview: this was done to understand if the output of the tool increased the awareness of the practitioners.

Architectural Smells Impact (RQ2.1) This part of the survey aimed at assessing the negative impact of the AS (Table in [22]). The question from 2.1 to 2.6 were adapted from the seven factors reported in [14]. In their paper, Martini et al. carry out an evaluation of these factors and report them as necessary and important for assessing the negative impact of Technical Debt. The assessment of the factors was adjusted, transforming the specific values used in [14] into a generic agreement Likert scale. Although this meant reducing the precision of the assessment, in this study we were interested in assessing a higher number of AS, and we therefore opted for a simplified set of questions. The factors and related questions in [22] are:

- **Reduced Development Speed** If the speed is reduced, the interest of the ATD is hindering evolvability and maintainability of the system (Q2.1)
- **Bugs related to the TD item** If many bugs are generated from the ATD, this greatly affects the maintainability of the system and the time wasted to fix bugs instead of developing new features (Q2.2).
- **Other qualities compromised** There are several qualities that can be affected by ATD, as shown in [7]. Here we used the ones suggested in the ISO standard [23] but excluding maintainability and evolvability (as they are already covered by the previous factors), (Q2.3).
- **Users affected** ATD might not involve a large part of the system, but can still affect a large number of developers. In this case, the extra cost of the interest would be multiplied by all the "victims" (Q2.4).
- **Frequency of the issue** The more frequent the negative impact occurs, the worse the interest (Q2.5).
- **Future growth of interest** To understand the overall negative impact of TD, it is important to assess the current negative impact (already covered by Q2.1-2.5), but we need to understand its future growth [7] as well (Q2.6).

⁶ For space reason, it is available at <https://drive.google.com/file/d/160TA9Q9jUIUpTBp-Wg7zu87e1Va3y1qr/view?usp=sharing>

- **Other extra costs** There might be other context-dependent extra costs to be considered when assessing the interest (Q2.7).
- **Spread in the system** The larger the portion of the system affected by the TD, the more ripple effects the interest might have on the organization and on the newly added code. This factor was already covered by the Arcan analysis and was included in the selection criteria according to the severity of the smell.

Finally, we asked an overall assessment of the negative impact associated with the AS (Q2.8). The correlation analysis supports the reliability of the chosen method. Such sanity check is explained with more details in section 4.2.

Architectural Smells Refactoring (RQ2.2) The second part focused on refactoring the architectural smell under analysis (Q3.1-3.4). First, we asked what strategy the practitioners would use to remove the smell (Q3.1). Then, we asked an estimation of how much time would be required to remove the smell (Q3.2). We agreed about the possible thresholds with the Chief Architect before the investigation. The thresholds were based on what the companies would consider *Low effort* and *High effort*. The next question aimed at understanding if the refactoring would create negative side effects (Q3.3). In fact, refactoring ATD might optimize one aspect of the system, but it could negatively affect other qualities [24].

Architectural Smells Prioritization (RQ2) We asked the overall priority that the practitioners would give to refactoring the ATD detected by the AS (Q3.4). In particular, we asked the practitioners to consider all the factors assessed so far, including questions 2.1-2.8 and 3.1-3.3. The aim was to understand how the negative impact and cost of refactoring affected the priority given by the practitioners.

Arcan Evaluation to identify ATD (RQ1) This part included general questions on Arcan, the detected smells and the use of software quality assessment tools during the development process, questions 4.1-4.7 (see Table [22]). The aim of these questions was to collect feedback from practitioners when detecting ATD by identifying architectural smells. In particular, we wanted to understand if the current state of the art on detecting AS would help the practitioners in managing their ATD. We asked for *a)* the difficulties emerged by using Arcan, *b)* the participants' past experiences using software analysis tools, *c)* quality index computation, *d)* estimate which are the most or least important type of architectural smells and *e)* evaluate if they are interested in using a new architectural debt index.

4 Results

Architectural Smells Selection Table 1 shows the AS found by Arcan in the four industrial projects. As we can see, we had a large amount of Unstable

Dependencies (UD). We found only two Hub-like Dependencies (HL), one in project A and one in project C. We also found many Cyclic Dependencies; more at class level (CCD) than at package level (PCD). Given the high amount of some smells (CD and UD) found in the projects, we had to select some AS instances from each category and for each project. For HL, we selected the two instances that we detected, while for UD, CCD and PCD we selected the instances with high severity according to Arcan. After this selection, we obtained the smells of Table 2a. Although Arcan shows the graph with the dependencies and the packages involved in a smell, it was difficult for the practitioners to easily navigate the results for the Package Cyclic Dependency smell, which were thus removed from the analysis.

4.1 Architectural Smells Identification (RQ1)

First, the practitioners assessed whether the AS was pointing to critical ATD issues, causing an increase of technical debt, or not. (Q1.1). In Table 2a, we can see the results of this first screening: on the "Discussed" column we report the number of smells that were initially discussed, while, on the "Problematic" column, we report the number of smells that were recognized as real problems.

From this analysis, we can see that **50% of the AS were considered problematic**, or else related to the presence of ATD. From the qualitative analysis, it is possible to better understand the reasons for the AS that were not considered related to ATD. One of the causes for this result was the kind of software that was assessed. For example, in project A, several Cyclic Dependencies were created by callbacks from anonymous classes in GUI components. When discussed with the practitioners, they clearly stated that "*On the Java server, you would never have this kind of stuff, but here it's kind of natural*". Callbacks for event listeners in the GUI components could not be easily replaced, and therefore the developers did not recognize CDs as problematic in those specific cases, but rather as a necessary solution. This leads to an interesting context-dependent finding: CDs might not need to be reported when analyzing GUI components.

The few HLs were very well understood and were recognized right away as issues, even without navigating the code.

On the other hand, UD were not easily understood by the practitioners: it was difficult for them to relate to the concept of Unstable Dependency, and they often disagreed on it being a problem. On the contrary, this kind of dependency was sometimes related to design patterns used on purpose. This is the case of an instance of Unstable Dependency in project A, that was caused by classes that applied a Strategy design pattern.

As a result of this first screening, we performed the subsequent assessment only on the AS listed as "Problematic" in Table 2a.

According to Question 1.2, only half of the problematic smells were known by the practitioners (Table 2a). This shows that the automatically reported AS improved the ATD awareness of the practitioners in 50% of the cases.

Fig. 1: Answers' for Questions 2.1 – 2.6 (Architectural Smells Impact)

4.2 Architectural Smells Prioritization (RQ2)

Architectural Smells Impact (RQ2.1) The answers to questions 2.1-2.6, are shown in Figure 1a. According to Question 2.1, in most of the cases the detected AS was associated with reduced development speed when adding functionalities, except for Unstable Dependencies: for this type of AS, practitioners seem to agree in 25% of the cases only.

As for Question 2.2, in more than 60% of the cases, the negative impact caused by Cyclic Dependencies is perceived as an increase of the number of bugs, while it does not happen for the other types of architectural smells.

Analyzing Question 2.3, some system qualities are negatively affected by the presence of Cyclic Dependencies in 60% of the cases and of Hub Like in 50%, while Unstable Dependency almost never affects them. Examples of other qualities affected by these ASs were mentioned in the interviews to be especially performance and testability.

According to Question 2.4, Unstable Dependencies seem to have an impact on technical debt that involves only few developers, while the impact seems to be higher for Hub-Like Dependencies (50% of the times) and Cyclic Dependencies (more than 60% of the cases).

As for Question 2.5, the developers perceive often the negative impact of Cyclic Dependencies in 60% of the cases, Hub Like Dependencies in all the cases and Unstable Dependencies in 50%.

According to Question 2.6, it is possible to assert that the impact will grow in the future, at least for every Hub-Like Dependencies and Cyclic Dependencies (75% of the times), but again it seems that we cannot assert the same for Unstable Dependencies.

In addition, Figure 1b shows an analysis of the answers for each type of architectural smell, where each column represents the degree of accordance for the statements related to the negative impact of each smell.

Table 2: Extracted architectural smells results and average answers by architectural smell given to questions Q2.8, Q3.2, Q3.3 and Q3.4

(a) Extracted Smells

Projects	Type of Smell	Discussed	Problematic	Known Yes	No
A	UD	2	0	0	0
	HL	1	1	1	0
	CD	4	1	1	0
B	UD	2	1	0	1
	HL	0	0	0	0
	CD	3	2	0	2
C	UD	1	0	0	0
	HL	1	1	1	0
	CD	3	3	2	1
D	UD	2	1	0	1
	HL	0	0	0	0
	CD	3	1	1	0
	Total	22	11	6	5

(b) Answers by smell

Smell ID	Average of			
	Q2.8	Q3.2	Q3.3	Q3.4
CD1	2	2	1	2
CD2	2	2	1	2
CD3	1	2	0	1
CD4	3	3.67	1.67	3
CD5	2.5	2.5	2.5	2
CD6	2.5	2.5	2.5	2
CD7	2.5	2.5	2.5	2
HUB1	1.5	2	0	1
HUB2	2	2	0	3
UD1	0	1	0	1
UD2	1	3	1.67	1
Total	1.90	2.48	1.29	1.86

Fig. 2: Answer's for questions 2.8, 3.2 – 3.4

According to Question 2.7, architectural smells can also have other impacts and generate problems for writing test cases or fixing conflicts during merging, and, in a few cases, the problems associated to some architectural smells instances, as for example for Cyclic Dependencies, were already identified and added as technical tasks to the backlog to be solved in the next future.

Finally, analyzing the answers to Question 2.8 regarding the overall negative impact, 42.9% Low Impact, 21.4% Medium-Low Impact and 28.6% Medium-High Impact. Moreover, in 7% of the answers (Figure 2a), the possible negative impact of the AS was not a problem (or negligible).

Architectural Smells Refactoring (RQ2.2) According to Question 3.1, in most of the cases the suggested refactoring requires the split of a class to reduce

the responsibility, sometimes to move some logic from a class to another, or to move some logic by creating a new class.

Analyzing Question 3.2, the time for refactoring AS is more than 8 man hours for 93% of the cases and in 14% of the cases, exceeds 100 man hours (Figure 2b Question 3.2). This was the case for example of a Cyclic Dependency identified in project D. Performing refactoring of this instance of AS, according to the practitioners, would involve reviewing the architecture of a main part of the system and reimplementing the logic revisiting the patterns used in this context.

According to Question 3.3 in 60% of the cases practitioners seem to agree that conducting refactoring would not create side effects.

According to Question 3.4, practitioners assigned *Low priority* to 100% of the UD instances, 50% to the HL and 25% to CD. They assigned higher priorities to the other instances of CD and HL (Figure 2d Question 3.4). The results on refactoring have been deeply investigated in the following section in order to understand the relationships between the aspects covered by the questions.

Fig. 3: Correlation analysis among the variables related to impact factors, overall negative impact, side effects, effort and refactoring priority. The white cells represent non-significant correlations (p-value > 0.05)

Correlation Analysis on the Prioritization of ATD In order to understand the relationships among cost, impact and overall prioritization of ATD, we ran Pearson correlation tests on the agreement scores. In particular, we compared questions 2.1-2.6 (impact factors) 2.8 (overall negative impact), and 3.2-3.4 (effort, refactoring side effects, and overall refactoring priority), shown in Figure 3. The scores in the white cells represent non-significant results (we set p-value < 0.05 for significance), so we discuss the red cells only.

First, we can see how there is a medium-strong and significant correlation between all the impact factors (2.1-2.6) and the overall negative impact. This confirms that all the chosen factors were contributing to the developers' perceived negative impact, but none of them seems redundant. This finding also confirms that the chosen method, to assess the negative impact

of ATD, can be considered sound.

Fig. 4: Answers' stacked charts for questions 4.1 – 4.7

We can observe how the refactoring effort has a strong correlation with the side effects of refactoring the AS. This makes sense, as the most side effects the refactoring would generate, the more costly it would be to remove the ATD.

The strongest correlation is between the priority of refactoring and the overall negative impact (0.88). This shows that the negative impact, perceived by the practitioners, is the main driver for the prioritization of refactoring. This means that, to prioritize ATD, practitioners need to know its impact (or else its interest). It is finally interesting to observe that there is a medium-strong *positive correlation* between the effort and the assigned priority. Although this is counterintuitive (more cost would suggest less priority), this finding implies that practitioners would prefer to refactor ATD that generates negative impact despite its higher cost of refactoring.

Architectural Smell Detection Evaluation (RQ1) According to Question 4.1, half of the practitioners found Arcan's output quite difficult to understand without using the graph generated by Neo4j. According to Question 4.2, 75% of the practitioners make use of software analysis tools during development activity: in particular, they used SonarQube (Figure 4a Question 4.2); for Question 4.3 all the practitioners expressed at least a *Mid-Low* interest in using Arcan as support during the development of a project (Figure 4a).

As for Question 4.4, 75 % of the practitioners found that Cyclic Dependency is the most important smell to detect, while (Question 4.5) Unstable Dependency seems to be the less important to detect (Figure 4d). Lastly (Question 4.6), 75% of the practitioners use an index to measure code quality (Figure 4a) and, according to Question 4.7, all the practitioners seemed interested in using an architectural debt index based on architectural smells detection (Figure 4a).

5 Discussion

We discuss the results in relation to each RQs.

RQ 1: How do AS help practitioners in identifying ATD? Given the results outlined in the previous sections, we can assert that AS detection increases awareness on ATD as 1) it highlights issues that should be considered as real problems but were not known according to practitioners and 2) in the few cases where the problem was known, it provided additional unbiased evidence of its presence.

RQ 2.1: How is the architectural smells' negative impact perceived by the practitioners? The results obtained in the analysis suggest that Unstable Dependency is the smell that causes less negative impact compared to the other smells considered in this analysis. In fact, practitioners do not consider this smell as a real problem in most of the cases. Hub-Like Dependency is the one that gives a higher perception of the negative impact and whose effect will grow worse in the future, since all the answers to the corresponding questions (2.6 and 2.8) are "agree" or "somewhat agree".

The smell that seems to be more impactful is Cyclic Dependency, although some of them were not considered real problems, meaning that they don't represent technical debt according to the practitioners.

In conclusion, the detection of these two smells helps practitioners. While the few HL Dependencies were considered a problem, CD need to be better filtered before being a reliable indicator of a serious presence of ATD.

RQ 2.2: What is the refactoring cost of AS? which kind of AS cost more to remove? Cyclic Dependency is the smell that requires more time to be refactored, and the one that creates more side effects during refactoring. For every type of smell, refactoring requires several hours to be performed, in some cases, for Cyclic Dependencies, more 100 man hours. On the contrary, Hub-Like Dependency is the architectural smell that requires a Mid-Low number of hours for refactoring without having any side effect.

RQ 2: How do practitioners prioritize ATD revealed through AS? Combining the results of RQ2.1 and RQ2.2, we can say that Hub-Like Dependencies seem to be the most convenient smell to detect and to refactor (best ratio cost/benefits). Cyclic Dependencies are also important, but need to be better filtered, because some are considered as a necessary coupling, and some have a really high cost of refactoring and therefore it is not clear if they should be prioritized for their removal or not. Finally, Unstable Dependencies are perhaps the less useful smell to detect and refactor.

6 Limitations and Threats to Validity

Validity threats for case studies are proposed in [13].

As for construct validity, there is a possibility that the practitioners misinterpreted what the AS represents or what we asked in the questionnaire. However, we thoroughly mitigated these threats as explained in the research design, with previous workshops explaining the AS.

As for internal validity, it is unlikely that the negative impact reported by the practitioners would be affected by factors that are not the AS, as we were

careful to inquire the main causes of negative impact when investigating the code. The correlation analysis do not imply causality per se: however, this threat is mitigated by the construction of the questionnaire and by further qualitative evidence collected, which supports causality.

There is a threat to external validity. The case-study has been conducted in a single organization, although we selected four quite different projects, and a limited amount of AS were in-depth studied.

As for reliability, two researchers were present during the investigation, while the results were checked by multiple researchers. Furthermore, the study is fully reproducible. The practitioners might have been biased when recognizing the negative impact of the code, due to peer pressure. However, they were capable of recognizing several harmful smells.

The results are based on practitioners' experience and perception. This means that the real cost, impact, and the consequent optimal prioritization of ATD might differ from the one reported here. However, practitioners are the ones suffering by ATD and are the final users of AS, so their perception is of utmost importance to understand how to manage ATD using AS.

7 Conclusion and Future Developments

In this paper, we performed a case study within a large software company, to understand how practitioners identify and prioritize Architectural Technical Debt using automatically detected architectural smells. Four industrial projects have been analyzed, and a sample of the detected AS has been thoroughly inspected by the projects' practitioners to find the causes of the issues, and to assess their negative impact and refactoring costs based on their perception and experience. We found which AS pointed at the most harmful Architectural Technical Debt (RQ1), which ones have more impact (RQ2.1), which ones cost more to refactor (RQ2.2) and which ones are, overall, more convenient to detect and prioritize (RQ2).

From the combined results, we can conclude that using AS to identify and prioritize ATD was considered useful: the tool helped identifying half of the problems that were not previously known by the practitioners, and provided evidence for the known ones. However, some AS were not considered high priority, which helps researchers to further improve and filter the automatically revealed AS. Cyclic Dependency was the AS with the worst impact but also the most expensive to refactor, while Hub Like Dependency has also a similar strong negative impact but seems to be the most convenient to detect and to refactor (less costly). On the contrary, Unstable Dependency was not perceived as an issue.

In the future, we plan to perform new case studies in other industrial domains and companies. We aim to better explore the refactoring cost of the smells to improve the prioritization of the smells to be removed first. According to this aspect we would like to ask practitioners to evaluate the usefulness of a new Architectural Debt Index [25], which allows to identify and assess the overall architectural debt of a project by taking into account the severity of each archi-

tectural smell. The index can be used to evaluate the most critical parts and to monitor the evolution of the architectural debt during the project history.

Finally, we aim to analyze and detect through Arcan other categories of architectural smells other than those related to dependency issues, such as smells related to the interface (Ambiguous Interface, Redundant Interface and Unstable Interface [26]) or smells related to performance or security issues.

8 Acknowledgments

The research leading to these results has received funding from the European Unions Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 712949 (TECNIOspring PLUS) and from the Agency for Business Competitiveness of the Government of Catalonia. We thank Chalmers University of Technology and University of Gothenburg for the support.

References

1. Besker, T., Martini, A., Bosch, J.: The Pricey Bill of Technical Debt: When and by Whom will it be Paid? In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). (September 2017) 13–23
2. Cast Software: Cast (2018) <http://www.castsoftware.com/Products>.
3. Headway Software Technologies: Structure101 (2018) <http://structure101.com/products/>.
4. hello2morrow: Sonargraph (2018) <https://www.hello2morrow.com/products/sonargraph>.
5. Arcelli Fontana, F., Pigazzini, I., Roveda, R., Tamburri, D.A., Zanoni, M., Nitto, E.D.: Arcan: A tool for architectural smells detection. In: Int'l Conf. Software Architecture (ICSA 2017) Workshops, Gothenburg, Sweden (April 2017) 282–285
6. Kazman, R., Cai, Y., Mo, R., Feng, Q., Xiao, L., Haziyevev, S., Fedak, V., Shapochka, A.: A case study in locating the architectural roots of technical debt. In: Proceedings of the 37th IEEE Intern. Conf. on Software Engineering (ICSE 2015). Volume 2. (2015) 179–188
7. Martini, A., Bosch, J.: An empirically developed method to aid decisions on architectural technical debt refactoring: Anacondebt. In: Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14–22, 2016 - Companion Volume. (2016) 31–40
8. Martini, A., Sikander, E., Madlani, N.: 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**(Supplement C) (January 2018) 264–279
9. Yamashita, A.F., Moonen, L.: Do developers care about code smells? an exploratory survey. In: 20th Working Conf. on Reverse Engineering, WCRE 2013, Germany, 2013. (2013) 242–251
10. Soh, Z., Yamashita, A., Khomh, F., Guéhéneuc, Y.: Do code smells impact the effort of different maintenance programming activities? In: IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14–18, 2016 - Volume 1. (2016) 393–402

11. Yamashita, A.: Assessing the capability of code smells to explain maintenance problems: an empirical study combining quantitative and qualitative data. *Empirical Software Engineering* **19**(4) (2014) 1111–1143
12. Palomba, F., Bavota, G., Penta, M.D., Oliveto, R., Lucia, A.D.: Do they really smell bad? A study on developers' perception of bad code smells. In: 30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014. (2014) 101–110
13. Runeson, P., Höst, M.: Guidelines for conducting and reporting case study research in software engineering. *Empirical Softw. Engg.* **14**(2) (April 2009) 131–164
14. Martini, A., Bosch, J.: The Magnificent Seven: Towards a Systematic Estimation of Technical Debt Interest. In: Proceedings of the XP2017 Scientific Workshops. XP '17, New York, NY, USA, ACM (2017) 7:1–7:5
15. Lippert, M., Roock, S.: Refactoring in Large Software Projects: Performing Complex Restructurings Successfully. Wiley (April 2006)
16. Macia, I., Arcoverde, R., Cirilo, E., Garcia, A., von Staa, A.: Supporting the identification of architecturally-relevant code anomalies. In: Proc. of the 28th IEEE Intern.Conf. on Soft. Maint.(ICSM 2012), Italy, IEEE (2012) 662–665
17. Arcelli Fontana, F., Pigazzini, I., Roveda, R., Zanoni, M.: Automatic detection of instability architectural smells. In: Proceedings of the 32nd International Conference on Software Maintenance and Evolution (ICSME 2016), Raleigh, North Carolina, USA, IEEE (October 2016) ERA Track.
18. Al-Mutawa, H.A., Dietrich, J., Marsland, S., McCartin, C.: On the shape of circular dependencies in java programs. In: Proc. 23rd Australian Software Engineering Conference (ASWEC 2014), Sydney, Australia, IEEE (April 2014) 48–57
19. Roveda, R.: Identifying and Evaluating Software Architecture Erosion. PhD thesis, Universit degli studi di Milano - Bicocca (3 2012)
20. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston, MA, USA (1999)
21. Lanza, M., Marinescu, R.: Object-Oriented Metrics in Practice. Springer-Verlag (2006)
22. ESSeRE Lab: Questionnaire table. Web site <https://drive.google.com/file/d/160TA9Q9jUIUpTBp-Wg7zu87e1Va3y1qr/view?usp=sharing> (2018)
23. ISO - International Organization for Standardization: System and software quality models. Web site (2015)
24. Arcelli Fontana, F., Roveda, R., Vittori, S., Metelli, A., Saldarini, S., Mazzei, F.: On evaluating the impact of the refactoring of architectural problems on software quality. In: Proceedings of the Scientific Workshop Proceedings of XP2016, Edinburgh, Scotland, UK, May 24, 2016. (2016) 21
25. Roveda, R., Arcelli Fontana, F., Pigazzini, I., Zanoni, M.: Towards an architectural debt index. In: Proceedings of the Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Technical Debt track, Prague, Czech Republic, IEEE (August 2018)
26. Garcia, J., Popescu, D., Edwards, G., Medvidovic, N.: Identifying architectural bad smells. In: CSMR 2009, Germany, IEEE (2009) 255–258