Received: 1 November 2015 | Revised: 23 February 2017 | Accepted: 19 March 2017

DOI: 10.1002/smr.1877

WILEY Journal of Software: Evolution and Process

RESEARCH ARTICLE

On the interest of architectural technical debt: Uncovering the contagious debt phenomenon

Antonio Martini | Jan Bosch

Chalmers University of Technology, Göteborg, Sweden

Correspondence
Antonio Martini, Chalmers University of Technology, Göteborg, Sweden.
Email: jan.bosch@chalmers.se

Abstract

A known problem in large software companies is to balance the prioritization of short-term and long-term business goals. As an example, architecture suboptimality (Architectural Technical Debt), incurred to deliver fast, might hinder future feature development. However, some technical debt generates more interest to be paid than other. We conducted a multi-phase, multiple-case embedded case study comprehending 9 sites at 6 large international software companies. We have investigated which architectural technical debt items generate more interest , how the interest occurs during software development and which costly extra-activities are triggered as a result. We presented a taxonomy of the most dangerous items identified during the qualitative investigation and a model of their effects that can be used for prioritization, for further investigation and as a quality model for extracting more precise and context-specific metrics. We found that some architectural technical debt items are contagious, causing the interest to be not only fixed, but potentially compound, which leads to the hidden growth of interest (possibly exponential). We found important factors to be monitored to refactor the debt before it becomes too costly. Instances of these phenomena need to be identified and stopped before the development reaches a crises.

KEYWORDS

agile software development, architectural technical debt, effort, multiple case study, qualitative model, sociotechnical phenomena

1 | INTRODUCTION

Large software industries strive to make their development processes fast and more responsive, minimizing the time between the identification of a customer need and the delivery of a solution. At the same time, the responsiveness in the short-term deliveries should not lead to less responsiveness in the long run.

To illustrate such a phenomenon, a financial metaphor has been coined, which compares the act of implementing suboptimal solutions, to meet short-term goals, to taking debt, which has to be repaid with interests in the long term. The term *technical debt* (TD) has been first coined at OOPSLA by W. Cunningham[1] to describe a situation in which developers take decisions that bring short-term benefits but cause long-term detriment of the software. The term has recently been further studied and elaborated in research. Tom et al[2] first mentioned architectural technical debt (ATD, categorized together with Design Debt). A further classification can be found in the work of Kruchten et al,[3] where ATD is regarded as the most challenging TD to be uncovered, as there is a lack of research and tool support in practice. ATD has been further recognized in a recent systematic mapping[4] on TD. Finally, a literature on ATD has been published recently.[5] Such recent body of knowledge highlights the lack of empirical research on ATD and especially on its interest, which motivates the investigation in this paper.

According to the glossary proposed in previous work,[6] we recall here that the ATD present in a system is a set of "suboptimal solutions." The suboptimality is considered with respect to an optimal architecture for supporting the business goals of the organization. A single architectural suboptimal solution is considered the debt and is called an *ATD item*. Each ATD item has a cost to be refactored, the principal, and an interest, or else an extra cost that is paid or is going to be paid in the future by the stakeholders because of the ATD item.[6]

An optimal solution refers to the architecture identified by the software and system architects as the optimal trade-off when considering the concerns collected from the different stakeholders (which is the "desired" architecture). An example of ATD might be the presence of structural violations.[7] In the rest of the paper, we call the suboptimal solutions *inconsistencies* between the implementation and the

architecture, or *violations*, when the optimal architecture is precisely expressed by rules (for example, for dependencies among specific components). However, such optimal trade-off might change over time because of business evolution and new information collected during implementation.[8] It is therefore difficult to anticipate all possible optimal solutions at the start of a large project. For this reason, it becomes important to continuously monitor the architecture to identify and refactor *costly* emerging suboptimality (ATD) rather than relying on upfront design *only*.

The costs related to ATD are based on two main concepts in TD literature: principal and interest. The principal is the cost of refactoring that is necessary to remove the debt. In a concrete example, if a system contains ATD in the form of a forbidden dependency between two components, the principal is the cost of reworking the system to eliminate the dependency. The interest is considered all the extra cost that is associated with the ATD; in other words, all those extra costs that would not be paid by the organization if the ATD would not be present in the system. For example, if the system is not well modularized, each change (for example, a bug fix) might require the developers to change several parts of the system, which would require more time and resources from the developers.[9] An ATD item is usually refactored if the principal is considered less than the interest left to be paid.[10-12] Estimating principal and interest of ATD is therefore one of the main goals of current research. However, such task is extremely difficult. In particular, the interest estimation might include several kinds of costs related to different activities. This paper is a first step toward calculating the *whole interest*. To do so, we connect the theoretical TD framework to concrete empirical cases.

A particularly important kind of interest is the so-called *compound interest*[13]; such interest does not only include a fixed amount of extra cost but causes the growth of available principal as well, which in turn causes more interest to be paid. Consequently, the interest might grow linearly but could also grow, according to the literature, exponentially.[13] It is therefore a very dangerous kind of interest. However, the existence of compound interest has been hypothesized as a theoretical concept, but no empirical evidence has been reported according to the literature review in previous work.[13] An empirical study on what compound interest is and how it is generated is therefore of considerable importance.

In this paper, we conduct an empirical study involving several organizations and several practitioners. The aim is to understand which ATD items are the most dangerous for accumulated interests. Moreover, it is important to understand what kind of interest is associated with the various ATD, both for their recognition during development and for the development of methods, tools, and measurements. In particular, we need to understand how compounded interest, the costliest interest, is accumulated in practice. We therefore answer the following research questions:

RQ1: What are the most dangerous classes of ATD in terms of generated interest?

RQ2: What kind of interest is triggered by different classes of ATD?

RQ3: Are there socio-technical anti-patterns causing the ATD and its interest to increase over time causing the accumulation of compounded interest?

RQ4: How can contagious debt and, therefore, the accumulation of compounded interest, be identified and stopped?

The main contributions of the paper are

- We have qualitatively developed and validated (through multiple sources) a taxonomy of effortful ATD classes.
- We link specific interests to specific classes of ATD by identifying recurring sociotechnical phenomena and consequent extra activities that could lead to extra costs
- We have conceptualized, defined, and found empirical evidence of, an important phenomena of ATD, *contagious debt*; this is related to the occurrence of sociotechnical chains of events (that we call *vicious cycles*) responsible for the increment of interest and that potentially lead to the accumulation of compound interest.

We have identified propagation factors, responsible for making the ATD contagious. By monitoring the growth of such factors, the practitioners can proactively avoid the accumulation of compound interest.

The rest of the paper is structured as follows: Section 2 gives the reader more references and background on ATD and on the conceptual framework used in this study. In Section 3 we explain our research design: overall design, description of the cases, methods for data collection, and analysis and evaluation of results. In Section 4 we list the results. In Section 5 we examine how the results address the RQs, discuss practical and theoretical implications of this study, and discuss the degree of validity of each result. We also point at limitations and open issues for future research and discuss the related work. We summarize the conclusions in Section 6.

2 | RESEARCH DESIGN

We conducted a 2-year long, multiple-case, embedded case study involving 9 Scandinavian sites in 6 large international software development companies. We decided to collect data from as many large companies as we had access to: this is to increase the degree of source triangulation[14] (collecting supporting evidence from different sources rather than from a single context) and therefore of generalization. The study has been conducted in an iterative fashion, as visible in Figure 1: first, we conducted a preliminary study investigating the high-level needs of the involved companies to understand their ATD situation (phase 1). Then, we have surveyed the practitioners to understand, more concretely, which ATD items were leading to costly interest (phase 2). The results were then evaluated, by conducting plenary sessions with many practitioners (phase 3) and by studying several cases in details (phase 4) to test and refine the previously developed theories and to add more evidence. This iterative process of investigation is well known as *systematic combining*.[15] A first theory is built and then refined as more evidence from the empirical world is collected. This iterative and continuous approach is also in line with the flexibility nature of case studies suggested in previous work.[14] This research process was chosen given the lack of previous empirical evidence in literature related to the research problem.

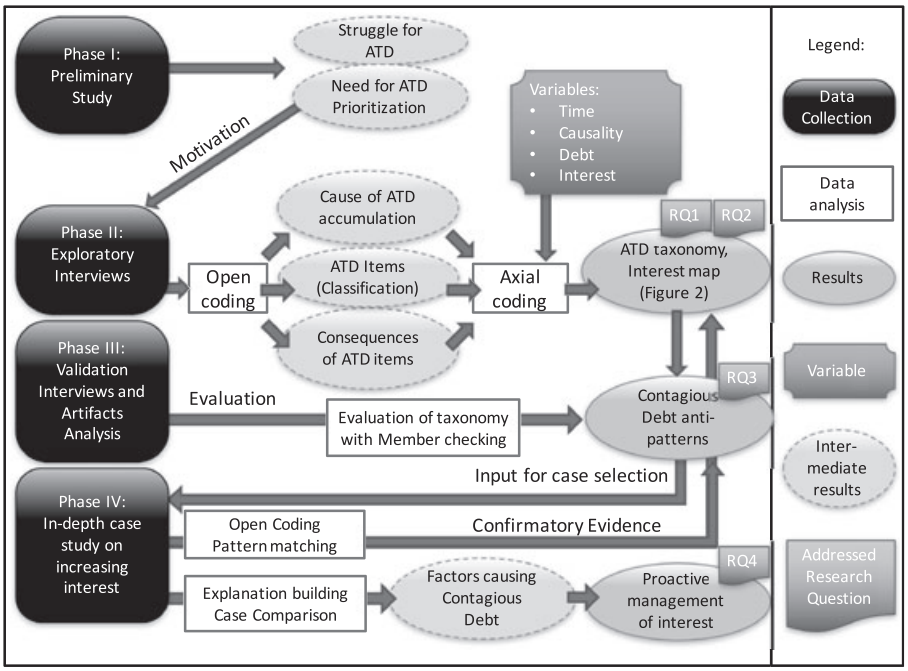

FIGURE 1 Our research design: data collection, inductive and deductive analysis, and results on the basis of different sources

2.1 | Case selection

We have used an embedded multiple case study,[14] where the unit of analysis is an (subpart of the) organization: the unit needed to be large enough, developing two or more subsystems involving at least 10 development teams. The total units studied were 9. We selected, following a literal replication approach,[16] 5 companies: A, B, C (4 subcases), D, and F large organizations developing software product lines, having adopted, on different maturity levels, agile software development, and had extensive inhouse embedded software development. We also selected company E, a "pure-software" development company, for theoretical replication[16] (hypothesizing different results from another companies).

Company A is involved in the automotive industry. The development in the studied department is mostly inhouse, recently moved to SCRUM. Company B is a manufacturer of recording devices. The company used SCRUM, has hardware-oriented projects, and use extensively open-source software. Company C is a manufacturer of telecommunication system product lines. They have long experience with SCRUM-based cross-functional teams. We involved 4 different departments within company C (C_1, C_2, C_3, and C_4). Company D used SCRUM to develop a product line of devices for the control of urban infrastructure. Company E is a "pure-software" company developing optimization solutions. The company has used SCRUM. Company F is developing devices used for defense systems. Some of the contextual factors relevant for the investigation are visible in Table 2.

2.2 | Data collection and analysis

The 4 phases (black boxes) of the investigation and their results are visible in Figure 1. The properties and the number of participants are also summarized in Table 1. For each phase, we describe the data collection procedure and the analysis that followed. All the interviews were recorded and transcribed. The analysis was done following approaches on the basis of Grounded Theory[18] and from case-study research,[14,16] frequently used for the analysis of large amount of semistructured, qualitative data, representing complex combinations of technical and social factors. For the analysis, we used a tool for qualitative analysis, (atlas.ti), which allows the categorization and

TABLE 1 Properties and Numbers Related to Data Collection

Phases of Data Collection	No. Participants	No. Sessions	Companies	Roles involved
Phase 1 (preliminary interviews)	25	3	Company-specific	Developers, architects, testers, line managers, and scrum masters
Phase 1 (evaluation workshop)	40	1	Cross-company	Developers, architects, and line managers
Phase 2 (group interviews)	26	6	Company-specific	Developers, architects, and product owners
Phase 3 (evaluation workshop)	10	2	Cross-company	Architects and line managers
Phase 3 (evaluation interviews 1)	10	1	Cross-company	Architects and product owners
Phase 3 (evaluation interviews 2)	12	1	Cross-company	Architects, developers and scrum masters
Phase 3 (validation workshop)	20	2	Cross-company	Architects
Phase 4 (in-depth cases analysis)	39	5	Company-specific	Architects and developers, (other stakeholders involved)
Informal interaction	7	NA	Company-specific	Software and system architects

analysis of qualitative data according to the approaches described in previous work[18] and keeps track of the links between the codes and the quotations they were grounded to: this was done to create a *chain of evidence*.[14,16] We have also used *pattern matching*[16] when analyzing the cases to validate the previous theories.

2.2.1 | Phase I—preliminary study

Data collection

We conducted a preliminary study involving 3 cases, in particular A, C1, and C2. We explored the needs and challenges of developing and maintaining architecture in an agile environment in the current companies. This phase contributed in identifying and selecting the research questions according to the studied industrial setting. We organized 3 multiple-participant interviews of about 4 hours at the different sites involving several roles: developers, testers, architects responsible for different levels of architecture (low-level patterns to high-level components), and product managers. The results from the first iteration were validated and discussed in a final one-day workshop involving 40 representatives from all the 9 cases (see Table 1).

In the preliminary study, we asked the following open questions:

- "How do you control consistency between the implementation and the architecture?"

This question was intended to understand to what extent ATD was managed in the companies. The difference between the implementation and the architecture represents the debt.

- "Which architecture risk management activities are you employing on different level of abstraction?"

This question was meant to investigate if the risk associated with the ATD interest were considered by the companies and how.

- "How do you prioritize architecture improvements?"

This question aimed at understanding what architecture practices were used in the organizations to prioritize ATD.

Data analysis

The data were analyzed in an explorative manner, using a technique called "open coding" analysis, suitable for exploratory studies, and we filtered the codes into "challenges." Then, by inductively categorizing the codes, we could see how several statements, consistently throughout all the cases, fell in the following categories:

- "reactive behavior to architecture drifting" (RBAD)
- "lack of continuous risk management activities for architecture drifting" (LCRMAD)
- "down-prioritization of architecture" (DPA).

The combination of these 3 categories leads to the phenomenon of accumulation of ATD; suboptimal solutions, the debt, are not evaluated continuously (RBAD), the risky effects are not understood in time (LCRMAD), and therefore, the refactoring is down-prioritized. The preliminary study showed therefore a major challenge in managing ATD. In particular, the studied companies emphasized the struggle, rather than in identifying the debt, in estimating its impact and therefore in prioritizing the items among themselves and comparing the ATD items against features (*need for ATD prioritization* in Figure 1), which led to the next phase.

2.2.2 | Phase II—investigating ATD classes of items and their effects (interest)

Data collection

In the second phase, we conducted 6 sets of interviews, one set for each company (Table 1). Each set lasted a minimum of 2 hours, and we included participants with different responsibilities, to cover many aspects including the source of ATD (developers), the architectural implications (architects and system engineers), the decisions taken after prioritization (product owners), and also the stakeholders of the effects (we included also testers and developers involved in maintenance projects when assigned to a dedicated project).

The formal interviews were also complemented with the preliminary study of software architecture documentation for each case, to which we could map the mentioned ATD items. The collaboration format allowed the researchers to conduct ad hoc consultations, several hours of individual and informal meetings with the chief architects (at least one per company) responsible for the documentation and the prioritization of ATD items.

Each set of interviews followed a process designed to identify ATD items with high interest. We took a retrospective approach, and we aimed at identifying real cases, occurred in the recent past, rather than rely on speculations about the future. We asked the following questions:

- "Can you describe a recent major refactoring, a high effort perceived during feature development or during maintenance work?"—This was asked to catch big ATD items refactored or high *interest* perceived for maintenance or slowed feature development.
- "Does such effort lead to architecture inconsistencies?"—We checked what kind of ATD was related to the interest.
- "What are the root causes for the identified architecture inconsistency?"—We investigated what caused ATD.

The output was a list of ATD items with large effort impact. The strength of this technique relies on finding the relevant architecture inconsistencies (ATD) by starting from the worst effects experienced by the practitioners instead of investigating a pool of all the possible inconsistencies and then selecting the relevant ones. From this first survey, we aimed at understanding more about how ATD was related to interest and to create a first theory.

Data analysis

This phase was exploratory, so we first analyzed the data in search for emergent concepts following the *open coding* technique,[18] which would bring novel insights on the analyzed issue. We coded the identified ATD

items in a taxonomy (*ATD items classification* in Figure 1). We used the same technique for identifying the key effect phenomena (*effects of ATD items* in Figure 1) and the causing factors that were related to each item (*factors for accumulation of ATD* in Figure 1).

We then apply an *axial coding* approach[18]; the codes and categories were compared to highlight connections orthogonal to the previously developed categories. Such analysis showed which category of items (*ATD items classification* in Figure 1) was connected to which effects (*effects of ATD items* in Figure 1). This way we could build an initial version of the *interest map* visible in Figure 2, in which we could represent the *debt* and the *interest* discovered through our investigation. The initial map is also visible in a previous study.[19]

2.2.3 | Phase III—evaluation interviews and artifact analysis

Data collection

The third phase consisted of two validation activities, and we organized 3 multiple-company group interviews, including all the roles involved in the investigation—developers, architects, and product owners. We showed the models for their recognition and improvement. In particular, we showed an early taxonomy of the map reported in Figure 2, and we showed the model of *contagious debt*. In this phase we collected confirmatory data, using the *member checking* technique,[14] about the phenomena and the models built during the previous phase. We asked, when the participants recognized the proposed models, to strengthen the evidence with further concrete and real examples. To test the completeness of the data, we also asked, where available, the analysis of artifacts such as lists of *technical issues* or *architectural improvements* identified within the company to understand if the identified items were mapped to the developed taxonomy. Such deductive procedure strengthened the inductive process used in the first and second phases. As a further validation step, we organized two plenary workshops with around 20 architects, in which participated employees from two other large companies not previously participating in the study, to further strengthen the results. In the workshops, we presented the findings, we asked if the participants agreed with the models and if they could provide cases to validate the models.

Data analysis

We went through the recordings of the 3 group interviews to understand if the participants agreed with the models. The comments and considerations from the participants were positive, and they mentioned further cases where the recognized phenomena were happening. We did not receive any negative feedback from practitioners standing against the models (for example, on the existence of *contagious debt*); this contributed to preliminary validate the results. However, given the threat of confirmation bias related to this approach, we decided to collect in-depth studies to validate the models with harder evidence.

2.2.4 | Phase IV—in-depth study of increasing interest

To collect stronger evidence on the previously gathered results, the map in Figure 2 and the model of *contagious debt*, we decided to follow up with another multiple case study. We wanted to verify the taxonomy of ATD classes and their interest and to find concrete cases to better understand the phenomenon of contagious debt. Such model suggested that, for some ATD items, the presence of interest was continuously increasing together with the principal, which can be considered as *compound interest*, a particularly dangerous form of ATD. We thus collected

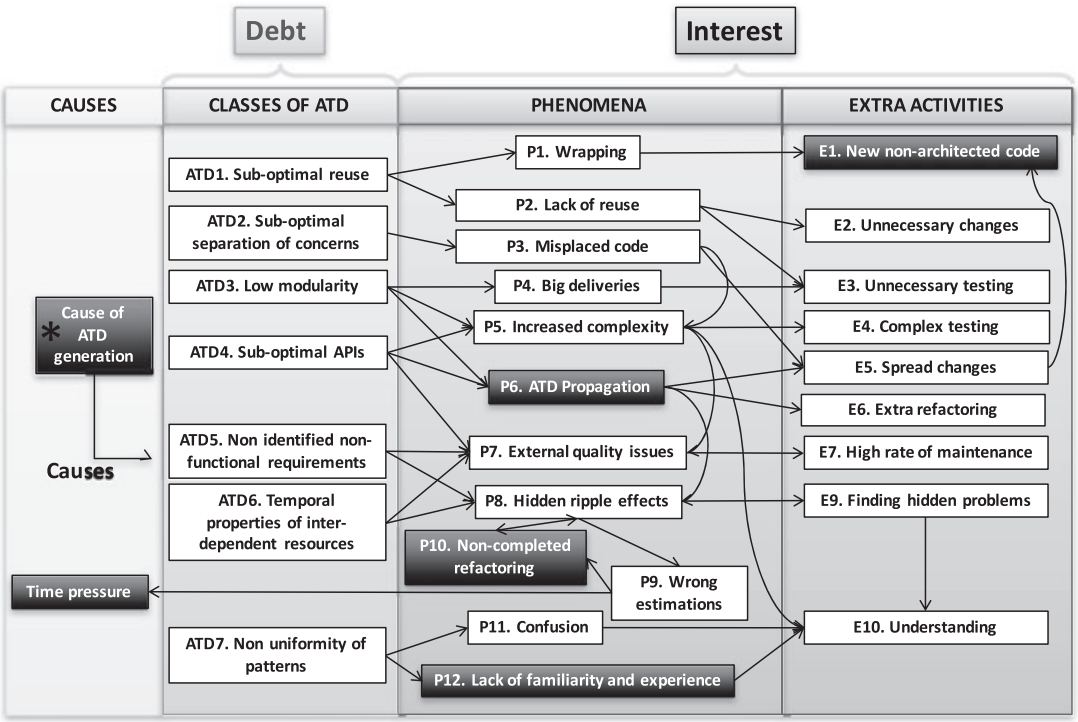

FIGURE 2 The model shows the causes for ATD accumulation (black boxes), the classes of ATD (which represent the debt), the phenomena that are caused by the items, and the final activities (which together represent the interest to be paid)

evidence from several cases (Table 2) on how the interest was growing, and if and how it was becoming compound. In this investigation, we observed how the architects were analyzing the cases, and we asked specific questions related to the evolution of principal and interest.

Data collection

The investigation was structured in the following steps (for which we developed and maintained an interview protocol, as recommended in previous work[14]).

- *Preliminary workshop.* We performed, at each studied site, an initial investigation with the purpose of identifying suitable cases with known or suspected high increment of interest. We performed this activity with the architects, and we used the model developed in phase 2 and evaluated in phase 3 to elicit suitable cases that seemed to be *contagious* (see arrow of *input for case selection* in Figure 1). This part also helped in minimizing possible construct validity threats, by aligning the concepts and the characteristics of the cases among the companies. We aimed at applying a replication strategy[16]: similar cases from the point of view of the main studied phenomenon, but slightly different from each other, to maximize the variance. This preliminary investigation lasted between 30 minutes and 1 hour, and it helped the identification of suitable participants able to provide useful information on the cases. For example, we selected the developers involved during the development, to understand how the ATD item was injected in the system and how it became contagious, but we also included developers, testers, and other stakeholders involved in the effortful evolution and maintenance of the studied ATD items.
- *Analysis of ATD case.* We set up a workshop with the participants in which we analyzed the ATD case in depth. We first asked for a high-level explanation of the issue, and the participants provided a description of the ATD item A short description of such issue which is reported in Table 2. Then, we performed a structured interview on the basis of the following:
  - $\mathbf{T_{now}}$ analysis. We performed an analysis of the current situation for the ATD item, and we investigated the *source* of the problem (the difference between the architecture and the implementation), the *current principal* (the estimated cost of refactoring such ATD item), and the *current interest* (extra cost estimated between now and a time far in the future).
  - $\mathbf{T_{past}}$ analysis. We asked how the ATD was *injected* in the system and how that happened. We then focused on the *propagation of the ATD*. It was important to understand at what point the interest of the ATD started to grow. By identifying the time and the factors that led the ATD to be propagated, we aimed at identifying what to monitor in order to avoid the contagion (*factors for increasing interest* in Figure 1). We asked this question only if we understood that the ATD item was already propagated. In some cases, the propagation was still hypothetical, so we did not need to analyze previous propagation, but we focused on the *future propagated interest* (explained below in $\mathbf{T_{future}}$ analysis). In other cases, the ATD item was refactored in time because of the estimated growth.
  - $\mathbf{T_{future}}$ analysis. We asked the participants to estimate the growth of the impact of the ATD item and the growth of the refactoring. This was done to understand what was the practitioners' current perception of the factors that would lead to the increment of interest and principal.
  - Evaluation. We asked the participant to use the current estimations for the growth of interest and principal to decide for the refactoring.

Data analysis

In this phase, we coded the qualitative data starting with a deductive approach, to identify the specific categories for each cases, as described in the data collection script. In particular, we have used the following main categories:

- Description of the item
- Interest (current or predicted)
- Principal (current or predicted)
- Factors for Increment of Interest
- Factors for Increment of Principal

We then applied the *explanation building* analysis strategy defined in previous work[16]; the purpose is to define causal links by using a narrative approach (used by most existing case studies[16]). The results from such analysis are shown in Table 2.

First, we linked the results from this analysis to the items in the taxonomy developed in the previous phase. The original taxonomy was updated with the new and more in-depth data. For example, if new extra-cost activities were found, and the classes were better specified. The outcome of this process is visible in Table 2, where, for each case, we have identified elements in the taxonomy. Some of the elements in the taxonomy were not found in the concrete cases, but we expected this as we selected less cases than in the previous phase, where we were trying to collect a broader set of phenomena.

In this phase we also used the *pattern matching* analytical strategy recommended in previous work.[16] Such strategy is used to verify the existence of previously formulated patterns (hypotheses) after a new collection of data. In our cases, we used the models obtained by the data collection conducted during phases 2 and 3 as *pattern tests* to be used during phase 4 to match the pattern. This kind of evidence contributed the evaluation of the formulated models related to *contagious debt*. A particular application of this strategy is reported in Section 4.2). The main high-level pattern, used as a pattern test was the one emerging from the definition of contagious debt: we tested if the increment of interest would lead to new principal with new interest. This would identify an instance of *contagious debt* and, therefore, a source of *compound interest*.

3 | RESULTS: A TAXONOMY OF ATD CLASSES AND RELATED INTEREST

The ATD accumulated in the system is commonly represented by *items*, also according to recent studies on TD management.[20,21] The items collected during phases 2 and 4 were grouped in classes. Then,

TABLE 2 Cases of Analyzed ATD

ID: Case_B_1	Classes: ATD6	Phenomena: P7, P8	Extra activities: E3, E7, E9	Status: Refactored
Debt description		Interest description		NOTES
There was a lack of well-designed "communication mechanism" for different applications sharing a memory resource, which provoked nondeterministic behaviors.		Created quality issues (bugs) that affected the users (in this particular case, the end customers). The bugs were difficult to reproduce and to patch without fixing the overall design. Also, there was extra testing of around 20% because of this issue. Interest propagation Several new (external) applications were going to use the shared resource; for each new application, the principal would grow as well as the interest.		There was pressure to refactor the TD from the open-source software community (external).
ID: Case_B_2	**Classes: ATD5**	**Phenomena: P6, P7**	**Extra activities: E1, E5, E6, E7, E10**	**Status: Refactored**
Debt description		Interest description		NOTES
A data structure, used by many projects, was fixed quickly (not well designed) to increase performance for a single and urgent project. However, the data structure was used in the same way in other projects. Either the project would have been entirely maintained outside of the product line, or the suboptimality would spread.		Usage of data structure was counterintuitive, prone to bug, hindered scalability, and led to workarounds. Interest propagation If the solution was rolled out for all the other projects, it would have had a lot of ripple effects, because the data structure was used in several places.		There was pressure to refactor the TD from the open-source software community (external).
ID: Case_B_3	**Classes: ATD1, ATD5**	**Phenomena: P1, P2, P6**	**Extra activities: E1, E2, E6, E7**	**Status: Propagated**
Debt description		Interest description		NOTES
The old version of an external library was used because it was considered too difficult and not worthy to adopt the new one. The new library was not "supposed" to be evolved more when the decision was taken, but then the project was restarted, adding new features to the library that would be useful. Also, the old library was not supported anymore		Lack of library features useful for the development of new applications. Lack of support for maintenance from the OS community, hence extra maintenance. Interest propagation The effort required to change the library was grown since the first release of the new library (time of decision to not update). Several new updates and applications in the systems, using the library, now suffered the interest and would have to be refactored to introduce the new library with the new features.		Interest (extra cost) not quantifiable in extra activities: lack of features due to the ATD. Decision of nonrefactoring due to external communication problem with OS community.
ID: Case_$C1_1$	**Classes: ATD2, ATD3**	**Phenomena: P3, P4, P5, P6, P7, P8, P9, P10**	**Extra activities: E2, E3, E5, E6, E1, E7, E9, E10**	**Status: Partly refactored**
Debt description		Interest description		NOTES
A common component was not modularized optimally in the beginning because of uncertainty about its future use by newly developed applications. Some applications started using it. The component was refactored, but the applications already using were not refactored because of other customer-related priorities, while new applications used the new version.		Before the refactoring, the suboptimal separation of concern and modularization led to code being misplaced, propagated ATD to new applications, increased complexity, and hidden ripple effects and wrong estimation of effort needed for refactoring, which led to the refactoring being not completed. These phenomena triggered a number of extra activities. Interest propagation The refactoring was growing at least linearly with the number of application added and users of the common component's API. Once refactored, the growing stopped, but the double maintenance (for the two versions of components) remained.		A full description of this case is contained in previous work[17]
ID: Case_$C1_2$	**Classes: ATD3**	**Phenomena: P5, P6, P8, P9, P10**	**Extra activities: E1, E6, E7, E9**	**Status: Propagated**
Debt description		Interest description		NOTES
There is a nonallowed (and not known) dependency between two components. The team discovered the dependency during feature development, but could not remove it for lack of time reserved in the agile sprint. Instead, more software was written that strengthened the dependency and made the component even more tightly coupled.		Every time a new feature was added to one component, extra code (not anticipated) is needed to be developed. The spread changes were complex and required the deep knowledge of the dependent component. Interest propagation The new code increased the existing ATD and its interest.		This item was investigated together with the team. The awareness of architecture was quite lower than the architect's one

TABLE 2 (Continued)

ID: Case_C3$_1$	Classes: ATD1, ATD5	Phenomena: P1, P6, P7	Extra activities: E1, E6, E7	Status: Injecting
Debt description		Interest description		NOTES
A component is being reused and adapted to save time for the short-term delivery of a customer feature that would allow the customization of some functionalities for the user. However, a better version is planned to be coded, which would bring better NFRs, such as better scalability and reliability.		Every time a new feature is added to the customer-specific product, more refactoring will be required to change to the updated component with better NFRs. Interest propagation The new customer-specific features did increase the existing ATD and its interest, making more and more difficult for the component to be refactored to increase the NFRs.		The component was "adopted" from a previous product and old architecture
ID: Case_C3$_2$	**Classes: ATD1, ATD5**	**Phenomena: P1, P6, P7**	**Extra activities: E1, E6, E7**	**Status: Injecting**
Debt description		Interest description		NOTES
A mediation layer between several components and several applications has been previously coded without satisfying scalability requirements. The solution is currently working, but could be refactored to be scalable.		The interest is not paid currently. Interest propagation Several new applications are going to be affected by the scalability problem. Also, complexity would grow because of the suboptimal mediation layer, and it would be exposed to the new applications.		
ID: Case_D$_1$	**Classes: ATD1, ATD4**	**Phenomena: P1, P5, P6, P7**	**Extra activities: E1, E3, E4, E6, E7, E10**	**Status: Propagating**
Debt description		Interest description		NOTES
An interface also used by external customization of the product is growing suboptimal. The backward compatibility required by previous customers causes the ATD to grow as new features are added in a nonoptimal way. On the other hand, a new interface would require double maintenance and the lack of backward compatibility.		Given the long-lived products, the interface was not designed to support the new features. The growth of the interface to support the added features is consequently suboptimal, leading to complexity of parameters and difficult to test behavior, which leads to wasted time and defects. Interest propagation For every new feature the cost of refactoring grows linearly. The interest is also at least linear with respect to the number of features. For each feature, the complexity of the interface increases as well as the maintainability.		The products require a lot of low-level coding and optimization, which brings additional complexity to handle in the development.
ID: Case_D$_2$	**Classes: ATD4**	**Phenomena: P5, P6**	**Extra activities: E3, E4, E6, E7**	**Status: Propagating**
Debt description		Interest description		NOTES
An internal interface between the platform and the application layers has problematic designed: to decouple the layers, the usage of it is causing too much complexity. A large number of components are already using it, which will need to be refactored together with the interface.		After the development of some applications, the architects have realized that the interface is suboptimal, being too complex, slowing down feature development, and being difficult and complex to test. Interest propagation For every new application the cost of refactoring grows linearly. New people are going to be hired, and the TD is going to be propagated to their knowledge: they will have to be retrained with the new system if the refactoring is postponed. Because the complexity of the new applications is also growing as they add new features, the time to refactor will increases further.		The products require a lot of low-level coding and optimization, which brings additional complexity to handle in the development.
ID: Case_E$_1$	**Classes: ATD4**	**Phenomena: P6**	**Extra activities: E5, E6**	**Status: Injected**
Debt description		Interest description		NOTES
A database component does not provide a standard API, and an application is using the private API. More applications are being developed, and if the standard API is not developed soon, they will all use the private API. In case the database needs to be changed to give better support for the features (probable), its refactoring would be massive compare to changing it without changing a standard API.		No interest is paid at the moment. On the contrary, accessing the private API is cheaper for the development of the current application. Interest propagation Several new applications are going to be developed and connected with the database component. If the standard API is not in place the new application would access the private API, and it would become very costly to evolve the database component afterwards, because it would require to change all the accessing applications		This company develops nonembedded software

TABLE 2 (Continued)

ID: Case_F$_1$	Classes: ATD2	Phenomena: P3, P6	Extra activities: E2, E3, E5, E6	Status: Propagated
Debt description		Interest description	NOTES	
The separation of business logic and GUI was not implemented. Business logic was embedded in the dialogs of the UI.		The misplaced code causes spread changes, increased complexity, and therefore difficult to understand, quality issues and extra maintenance. Interest propagation Every time a new dialog is added to the system, the debt is propagated with the addition of new misplaced business, as there is no component modularized for this part of the business logic.		This problem is between architecture and design TD.

We show the debt, its interest, its increment the connection with the taxonomy.

Abbreviations: API, application program interface; ATD, architectural technical debt; GUI, graphical user interface; TD, technical debt.

the classes were connected with their interest. First, we explain such connections, then, we represent them graphically in Figure 2. In Table 2, we report a summary of the in-depth cases studied in phase 4. In the same table, we report the references to the ATD classes and interest displayed in Figure 2. For a better understanding, we divided the interest into *phenomena* (sociotechnical patterns that represent a recognizable event during software development) and their effects in terms of triggered extra-development *activities* (which can help creating measurements for the interest). The combination of Table 2 and Figure would show the full chain of evidence from the concrete cases to the theoretical results.

The ATD items identified during the investigation can be grouped into 7 categories, ATD1-7. In the following sections, we described each class, and we map them to phenomena (P1-12) and to extra activities (E1-10) generated by these phenomena.

ATD1. *Suboptimal reuse*

Reuse is a well-recognized strategy to reduce costs and to increase speed. There are two main problems related to reuse, which can be considered ATD:

- P2. The lack of reuse (when such strategy is possible) represents ATD. The presence of very similar code (if not identical) in different parts of the system, which was managed separately and was not grouped into a reused component, leads to paying interest in duplicated activities. Such duplicated activities lead to unnecessary changes (E2) and unnecessary testing (E3).
- P1. Another phenomenon related to reuse is called "*wrapping*"; during the interviews, for example, as mentioned by an architect from company B, this phenomenon was also called "glue code." Such code is needed to adapt the reused component to the new context with new requirements. Such code is usually unstructured and suboptimal, because it is not part of the architectural design but rather developed as a workaround to exploit reuse (E1). The consequences are not direct, but the risk is that, with the continuous evolution of the system around the reused component, such glue code would grow, becoming a substantial part of the (sub-)system containing non-architected code and therefore ATD (of different kinds). This is why the box is black: representing the consequence of wrapping can be considered the cause for new forms of ATD.

It is important to notice that some of the developers, from different companies, mentioned that the lack of reuse (citing interviewees, "copy and paste") is not always considered a bad solution, but in some cases, it can be convenient to copy and paste and evolve the new code. In such case, though, it seems better, to avoid the wrapping phenomenon, rearchitecting the component together with the new code to avoid propagation of ATD.

ATD2. *Suboptimal separation of concerns*

In this case, the code is not well allocated in the components combining the system. A typical example is a layered architecture where a generic business logic layer has nonallowed dependencies to

(eg, functions calling) to the user interface layer. Such code is then misplaced (P3), which causes the changes to be spread in the system (E5), requiring a larger and riskier change applied by the developers. Also, such ATD causes an increment of the complexity (P5). Increased complexity leads to several extra activities, and according to our cases, testing the modules and their behavior in a complex architecture is complex (E4), and a more complex system requires developers to spend more time in understanding. Also, we found that, and this is also well known in literature,[9] complexity increases the error-proneness of the system, which leads to external quality issues (P7).

ATD3. *Low modularity*

This category includes the items that are related to the presence of many architectural dependencies among the components. The components can be said to be not loosely coupled or cohesive enough. This has also been recently recognized as TD in a dedicated study in which a quantitative analysis of low modularity and its interest (in terms of extra bug fixing) has been performed.[9]

In case C_2, the interviewees mentioned a concrete example of this phenomenon: the large amount of dependencies among the components in one of the subsystems, caused, each time a new release involved a small change, testing the whole subsystem (*big deliveries* or P4 in Figure 2). Such event hinders agile practices such as continuous integration, in which high modularity of the system allows the fast test of small portions of the code (for example, a single or a small set of components). In the picture, we called this E3—*unnecessary testing*. Also, having many dependencies among the components increase complexity (P5) and increase the chances of existing TD spreading (P6).

ATD4. *Suboptimal APIs*

The application program interface (API) of a component is the point of access among different parts of the system or different systems. The suboptimal design of APIs or the misuse of them is considered ATD. As an example of the first class, we can consider methods that contain too many parameters or that are difficult to use because they return too generic or unstructured values, which need to be interpreted and might create confusion. As an instance of the second kind, components might call private APIs (that are not supposed to be called outside the component) instead of public and well-designed ones; this, in practice, makes APIs that are privatized and not well-designed, public.

The effects of these kinds of problems might lead to increased complexity (P5), and, because they are the access point among components, they are the possible gateway for the spread of ATD (P6). In fact, the more components access a suboptimal API, the more likely it is that the suboptimality would be propagated to the code using it: for example, developers might implement workarounds. Also, once the API is used by many components, it would become very costly to refactor the components, in case the API would be refactored or changed (which means, the cost of refactoring would grow, E6). Finally, suboptimal APIs are considered more prone to bugs (P7, external quality issues) by the developers, leading to extra maintenance work (E7).

ATD5. *Non-functional requirements (NFR)*

Some NFRs, such as performance, scalability, and signal reliability, need to be recognized early during the development and need to be tested. The ATD items represent the lack of an implementation that would assure the satisfaction of such requirements, but also the lack of mechanisms for them to be tested. Some cases were mentioned by the informants: for example, in case C, the lack of a fault handling mechanism, turned out to be very expensive when added afterwards. Case A reported frequent struggles with nonfunctional requirements regarding memory consumption and communication bus allocation. Case B mentioned the difficulties in anticipating the future use cases for a given feature.

The introduction of such debt causes a number of quality issues (P7), which are difficult to be tested. The informants argue that it was difficult to repair this kind of ATD afterwards, especially for requirements orthogonal to the architectural structure, when the changes would affect a big part of the system: in such case, it was difficult for the developers to know all the affected parts in advance (P8, hidden ripple effects); therefore, quantifying the change and estimating the cost of refactoring has always been reported as a challenge (P9, wrong estimation of effort). Consequently, teams experience additional time pressure when the estimations are wrong. Also, the hidden ripple effects might lead to developers not recognizing parts that cannot be refactored and therefore to the dangerous presence this code that is not completely refactored (P10). Non-completed refactoring, in turn, leads to ATD that is not known.

ATD6. *Temporal properties of interdependent resources*

Some resources might need to be accessed by different parts of the system. In these cases, the concurrent and non-deterministic interaction with the resource by different components might create hidden and unforeseen issues. This aspect is especially related to the temporal dimension; as a concrete example from company B, we mention the convention of having only synchronous calls to a certain component. However, one of the teams used (forbidden) asynchronous calls (which represents the ATD).

Having such violation brings several effects, and it creates a number of quality issues (P7) directly experienced by the customer, which triggers a high number of bugs to be fixed (and therefore time subtracted to the development of the product for new business value, E7). However, the worse danger of this problem is related to the temporal nature of it. It is difficult, in practice, to be tested with static analysis tools. Also, once introduced and creating a number of issues, it might be very difficult for the developers to find it or to understand that such issue is the source of the experienced problems (explicitly mentioned by companies B, C, and D). A developer from site D mentioned a case in which an ATD item of this kind remained completely hidden until the use case slightly changed during the development of a new feature. The team interacting with such ATD item spent quite some time figuring out issues rising with such suboptimal solution. The *hidden* nature of these ATD and their ripple effects (P8) create a number of other connected effects.

ATD7. *Non-uniformity of patterns*

This category comprehends patterns and policies that are not kept consistent throughout the system. For example, different name conventions applied in different parts of the system. Another example is the presence of different design or architectural patterns used to implement similar functionalities. As a concrete example, in Case A, different components (ECUs in the automotive domain) communicated through different patterns.

The effects that are caused by the presence of non-uniform policies and patterns are of two kinds: the time spent by the developers in understanding parts of the system that are not familiar with (P11) and by understanding (E10) which pattern to use in similar situations (it causes confusion, P11). For example, in case A, the developers experienced difficulties in choosing a pattern when implementing new communication links among the components, because they had different examples in the code.

A phenomenon involved the feature teams interviewed at company C. In such context, the teams were unlinked from the architectural structure (each team could change any component necessary for developing a feature). The interviewees mentioned that the lack of experience and familiarity with the code favored the introduction of additional ATD: for example, a developer from company C mentioned that he applied a similar pattern found in the same component for developing a new feature. Unfortunately, such pattern was already ATD (it was not an optimal solution), and therefore, the developer increased the ATD.

4 | RESULTS: THE SOCIOTECHNICAL PHENOMENON OF CONTAGIOUS DEBT

In the previous section, we have described the connection between classes of ATD items with phenomena that might be considered dangerous for their costs when they occur during the development. Such cost represents the interest of the debt, and the previously explained categories of debt have been associated with a high interest to be paid. However, such associations can be considered as *fixed*, which means that each item brings a constant interest cost.[13] However, our analysis of the relationships among the different phenomena have brought to light chains of sociotechnical cause-effect relationships that cause the ATD to spread in a contagious manner: we call the spreading action also "*propagation*." We therefore define *contagious debt* as follows:

> **Definition 1.** *"A chain of events in which the consequences of an existing ATD item lead to the accumulation of new ATD in the system. The new ATD can be of the same kind or different, and it also generates new interest."*

This is a concrete definition that matches the abstract one for *compound interest* reported in previous work.[13] Contagious debt and the related compound interest are interesting phenomena; if the existing ATD is not repaid, this does not only create a *fixed* or *simple interest*,[13] but it leads to the creation of more ATD and therefore to more interest. The difference between the *simple interest* and the interest created by the contagious debt is explained in the equations below. In the first case, we have that

$$ATD \xrightarrow{Generates} Simple\ Interest \tag{1}$$

For example, according to the taxonomy in Figure 2, ATD1 generates simple interest in the form of E3. However, when the interest of ATD generates new ATD (by propagation), we have a chain of events of the kind:

$$ATD \xrightarrow{Generates} Simple\ Interest + \left(ATD \xrightarrow{Generates} Simple\ Interest + \left(ATD \xrightarrow{Generates} \ldots\right)\right) \tag{2}$$

For example, according to the taxonomy in Figure 2, ATD2 causes P3, which generates E5 (simple interest). However, E5 could lead to E1, which in turn can generate new ATD (of any kind). The new ATD then generates more interest, which could be simple or not. In the second case, the interest becomes compound. In general, instances of contagious debt are visible in our model in Figure 2 in the form of paths; the column on the left includes the possible causes of ATD accumulation (see the work of Martini et al[22] for details), which we have represented with black boxes (*cause of ATD generation*). We do not report all the causes of ATD in this paper, but they can be found in previous work.[8] Among the phenomena triggered by some classes of ATD, we can find also causes of ATD (also represented by black boxes). Analyzing Figure 2, a chain of events starts with a cause (all ATD items have a cause), then it involves ATD items and ends in a phenomenon that is *also* a black box: such black box will cause the creation of additional ATD and therefore of additional interest. Such chains of sociotechnical phenomena represent instances of contagious debt, according to our definition.

Also, we need to consider the following case: if all ATD items generate two new ATD items by propagation, the accumulation of ATD (and therefore the interest paid) results as nonlinear. Although contagious debt does not necessarily imply compound interest (the chain might not be continuous), contagious debt is a concrete mechanism that could potentially lead to compound interest: it is therefore noteworthy, as it differs from having ATD that does not propagate and creates simple interest. This is in line with previous work,[13] which states that compound interest might grow linearly but could potentially be accumulated in an exponential fashion. Although the concept of compound interest has been formulated theoretically, there is no empirical evidence of such kind of TD existing in practice for ATD or how it is

manifested. We present here the contagious debt phenomenon and concrete instances of such phenomenon, which potentially leads to the accumulation of compound interest.

We report below some of the instances of contagious debt from the cases. We cannot report all the cases in details, but in Table 2 we have added a column with the summary of the various instances of contagious debt.

1. *Glue code—propagation by reuse*

One instance of contagious debt was already explained for ATD1. In such case, suboptimal reuse triggered the creation of nonarchitected code; the growth of the system with glue code would generate more nonarchitected code (it is unlikely that new code added to nonarchitected code would be well designed). It is possible to see this behavior in several cases in Table 2, such as Case_D_1, where the reuse of an old legacy system led to the systematic increment of suboptimal interfaces that were spread to the new applications.

2. *Propagation by APIs*

This is one of the most recurring problems in the cases reported in Table 2. The API is usually the gateway for the ATD to spread. Being this one of the most mentioned phenomena, we describe in details of the cases (Case_E_1) in detail. In this case, there was a database in a layered architecture (we will refer to it as "component A") that was meant to not be directly accessed by other components. The reason for such architectural rule was that the company, in its long-term roadmap, saw the possibility of replacing the database with a "better" version or even with a different persistency mechanism that would have allowed the development of new features. However, in practice, the database component lacked a standardized interface; this difference between the desired architecture and the actual implementation was suboptimal, and therefore, it represented the resulting ATD item. In the system, there was also another component (we will call it "B") accessing component A (the database); component B would interact with A using the nonstandard, private interface provided by A. This meant that, even if the ATD item was located in component A, the debt spread into component B as B grew and other parts of the code needed to access A. This made the cost of removing the original ATD in component A higher, because component B also needed to be changed in many places. At the moment of investigation, the company needed to add other components (C, D, etc) that would allow the development of customized features for different customers. The new components (C, D, etc) had to interact with the database (component A); at this point, the company faced the decision of removing the ATD item before "spreading" itself to the new components, or implementing the new components *with* the ATD. However, the removal of the ATD item now would involve the refactoring, not only of the database (component A containing the ATD), but also of component B, which was interacting with A; clearly, the cost was much higher than changing only A. This was considered an interest of development delay when evolving the system. On the other hand, not removing the ATD item, would have meant spread new ATD to component C, D, and others making its removal even more costly and increasing the future interest to be paid.

From the concrete cases such the one explained above, we built the model shown in Figure 3; at a certain time T_0, the system contains an ATD item in component A. We consider this ATD as the original ATD item, for which the cost of removal is fixed (the *principal* P). At this point, there is no interest to be paid. At time T_1, there might be two (or more) events that contribute to the increment of the interest; the code grows around the original ATD, interacting with it, and another component needs to interact with comp A, causing the ATD to be propagated to comp B. We call these two quantities SWG_{P1} (software growth related to the principal at time 1) and $PropATD_{B1}$ (ATD propagated to component B at time 1). Now, the interest is $I = SWG_{P1} + PropATD_{B1}$. At time T_2, as the software grows again and a new component is added, we have 3 more quantities, SWG_{P2} (assuming that the software in comp A grows again), SWG_{B2} (growth of software in comp B), and $PropATD_{C2}$ (the ATD is propagated to a new comp C). The interest would therefore be $I = SWG_{P1} + PropATD_{B1} + SWG_{P2} + SWG_{B2} + PropATD_{C1}$. By assigning a fictitious value of 1 (for the sake of simplicity), and calculating the overall cost as P + I (Principal plus Interest), we have that at T_0 C = 1, at T_1 C = 3 and at T_2 C = 6. Each time the ATD is propagated, the growing code needs to interact with it, increasing the cost of its removal and propagating ATD even further.

Let us show what happens if we apply this model to the previous concrete case (Case_E_1). At T_0, when the component was created with

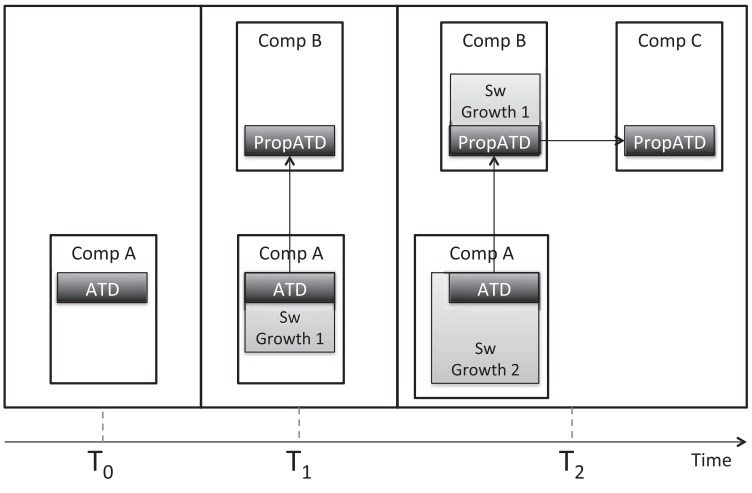

FIGURE 3 Model for contagious debt accumulation at different points in time: T_1, T_2, and T_3

the ATD, the cost of removing/not introducing the ATD item would have been C_0 = 1. At time T_1, when component B was already introduced, the cost would have been C_1 = 4 (removing the ATD from A, from the code around it, from B and from the code around B). At time T_2, after the new components C and D would have been introduced and grown and the ATD would have spread, the cost could have reached C_2 = 10 (the previous cost, C_1 = 4, plus the code grown around A and B, which is 2, plus the ATD introduced in C and D and the code around them, which is 4). This scenario implies that the components grow constantly, making it a worst-case scenario, but we have to consider also more components could have been added.

The cases studied suggest the steep growth of the interest on the principal over time with respect to the growth of the connected parts. It becomes of utmost importance, for the company facing the situation of being at T_0 or T_1, to know the risk of letting the ATD spreading around before the vicious circle has gone too far (T_2); at such point, the company could be in the situation of facing the choice of refactoring component A (which has become extremely costly), or renouncing to implement the new features connected with such change. The main goal becomes, at this point, to be able to estimate correctly how much the system is going to grow around the source of an ATD item.

3. *Propagation by unawereness*

The unawareness of hidden ATD, by the developers and architects, causes them to be unable to estimate correctly the time that is necessary to refactor another ATD item or to deal with it. Consequently, when a refactoring is planned to be performed in a certain time, it might result in being incomplete because of the wrong estimation due to the unawareness. Another case occurs when new features are added to the part of the system containing ATD: the time for delivering such features might increase. Incomplete refactoring has been recognized to be a cause for new ATD accumulation in the works of Martini et al[22] and Mamun et al[23] (in the latter one just for TD). To make things worse, the combination of wrong estimation leads to increased time pressure during development or refactoring, which in turn increases the probability of ATD accumulation.

To better describe this phenomenon, we refer to a concrete case reported by the informants during phase 2 (not included in Table 2); the architects identified an ATD item consisting of a violation, for which 3 different patterns were used to communicate among components in different parts of the system. Such problem had shown to create difficulties during development, because developers felt confused about when using one pattern or another. Therefore, a refactoring was planned by the architects to remove the 3 different protocols and replace them with a fourth one, which was intended to be kept consistent for the whole system. However, during the refactoring, several ripple effects where discovered, connected to the implementation of the three patterns to be removed. Such effects were not considered during estimation time. Given the time pressure to complete the refactoring, the result was the presence of a fourth protocol included in the system without the developers being able to remove the other three. In addition, the management would not prioritize such refactoring again, given that the problem was meant to be solved by the intervention. This example clearly shows how the presence of hidden ATD would generate more ATD in the system, making it contagious and creating a trend of continuous increment of ATD and interest to be paid.

4. *Propagation by complexity*

Although the complexity can be regarded itself as TD, in this case, we take the perspective of it being a factor for the growth of the principal related to another ATD item. This propagation factor was clearly mentioned in case Case_D_2; in such case, the architects discovered the ATD while new applications were developed. Consequently, when they had the option to repay the ATD, the applications were not fully developed, and most of them were not dependent on each other. However, the more the system would grow, with more interdependent applications, the more complex the code would become be to refactored. This would increase the refactoring cost, and therefore, the principal would grow together with the interest that was already discovered.

5. *Interest in testware and its propagation*

We found that in many of the analyzed cases, the interest is related to the testware (extra activity E_3 and E_4); often, unnecessary tests are performed because of the ATD (for example, in case Case_B_1, 20% more testing was estimated to be wasted in finding bugs related to the shared memory issue). This is important when measuring interest; no studies have so far related the interest to the extra effort reported in testing because of ATD. Another important point is that refactoring ATD implies, often, the refactoring of the testware as well. Monitoring the growth of the testware in relationship to code affected by ATD is therefore important to understand if the interest is going to increase over time. Especially in embedded software, for example, for safety critical systems, the testware might be substantial, and therefore, the interest on testing should be taken in consideration as one of the major factors of propagating ATD and increasing its interest.

6. *Propagation by new users*

A component might interact with a number of other components, applications (or features), or even with external systems. For example, in several cases, the increment of interest was considered linear to the number of features included in the roadmap. In those cases, the teams developing new software, which was interacting with the component containing ATD, were affected by the interest generated by the ATD. If the number of users affected by the ATD grows, the overall interest grows as well. It is therefore possible to use the information regarding the number of users interacting with the ATD to decide to refactor or not an item before the number of such users would grow. Another reason why the number of users can be important to calculate the interest is clear in case D_2, where the training of the newcomers would be more effective if it would happen after the refactoring; this way, the time for learning was limited to the new system only instead of to both systems. Furthermore, in the cases related to companies B and C2, the users were external to the organizational boundaries, ie, belonging to other organizations in the ecosystem. In such cases, the ATD would affect an undefined (and potentially high) number of users; such information constitutes a risk factor that needs to be taken in consideration

when calculating the interest. This is clear from the data collected in the case studied; in all the cases, where external users were going to be affected by ATD, it was decided to refactor the ATD item. This was a high priority even if in Case_C2$_1$, even if the refactoring was not completed, which caused an internal extra cost, the refactoring avoided the external users experiencing the interest of the ATD.

7. *Propagation of interest to other system qualities*

In many of the cases, the extra costs generated by the interest were not only related to maintainability but also estimated to worsen other system qualities. For example, if a component created performance, reliability, or scalability issues, when reused, in a new product, it might cause the new product to suffer from the same issues, as highlighted in cases Case_C3$_1$ and Case_C3$_2$. It is therefore crucial to monitor how ATD is going to affect the new system with respect to several system qualities important for the organization.

8. *Propagation by bad example*

As highlighted in Figure 2, the lack of uniformity of polices and patterns, combined with the Agile practice of having teams modifying also part of the system for which they are not familiar, leads to additional accumulation of ATD. Although this chain of events might not create continuous growth of interest, it is worth highlighting that this particular combination *might* lead to a vicious cycle in the worst cases. However, it also shows how ATD and its interest, in such situation, might increase more than it is perceived intuitively.

This phenomenon is concretely explained by a case involving feature teams interviewed at company C (see also the case Case_C1$_2$ in Table 2). In such context, the teams were not owning a specific part of the system (each team could "touch" any component necessary for developing a given feature). The interviewees mentioned that the lack of experience and familiarity with the code favored the introduction of additional ATD: for example, a developer from company C mentioned that he applied a similar pattern found in the same component for developing a new feature. Unfortunately, such pattern already contained ATD, and therefore, the developer increased the ATD. It is important to notice that such pattern might be repeated over and over again. In the studied case, the same violation of having nonallowed dependencies led to a high number of nonallowed dependencies in place, which represented a high lack of modularity in the system, for which a lot of interest was paid.

5 | DISCUSSION

5.1 | Toward a solution to timely refactor contagious ATD: monitoring propagation

The rationale for refactoring ATD is based on the evaluation of the cost of the whole refactoring versus the remaining cost of the interest.[12] As we have often found in the analyzed cases, when the ATD item was discovered, the cost and probability of the interest was not known or very uncertain. The same held for the propagation factors, they were usually unknown or very uncertain. Therefore, at such point in time, the ATD was not considered convenient by the participants to be avoided or refactored. Later on, when the interest became visible, the participants considered the refactored of the ATD item convenient. However, at such time, the principal was already high, because, over time, the ATD had spread in a contagious way. This caused a struggle in deciding to perform a refactoring.

In summary, when the interest becomes enough to justify the refactoring, it is usually too late to do so, because the refactoring is too costly with respect to the remaining interests to be paid. But then, when should we refactor? The answer seems to be related to the time variable and to the propagation factors:

> **Proposition 1.** *Developers and architects need to estimate the growth of interest before the principal has grown too much.*

We therefore identified the following main approaches necessary to proactively act to avoid the interest of ATD:

- *Iterative analysis*. The ATD needs to be identified and reevaluated periodically; this is necessary to understand if the interest is going to grow. The uncertainty on the future interest is similar to the uncertainty affecting feature development, and therefore, an iterative approach can help understanding the interest better as time goes.
- *Holistic calculation of interest*. Calculating the interest is challenging, but it is necessary to understand which ATD item should be avoided or repaid. However, no tools are currently available for calculating the interest, which makes the organization blind to a key factor affecting software development. Another important finding from this study is that the decisions usually need to take in consideration *all* the interest, which, as we can see in Figure 2, can be of various kinds. We provide a map in Figure 2 that can help, during the monitoring sessions, in understanding what phenomena are happening and what extra activities can be observed to understand the interest.
- *Contagious debt propagation identification*. Contagious Debt is a recurring high-level phenomenon leading to increasing interest, potentially generating nonlinear compound interest. Such phenomenon is caused by the propagation of the ATD and interest through several propagation factors and sociotechnical patterns. We provide a list of the factors that can be monitored to catch the contagiousness of a given ATD item, and we report several examples from real-world showing such sociotechnical patterns: this can be very valuable for organizations experiencing similar problems. A monitoring method, on the basis of these findings, has been reported as improving the estimation power of the architects.[12] The factors reported here can be used to develop further methods and tools.

5.2 | Summary of contributions for each research question

The investigation that we have performed contributes to address the research questions with the following results:

RQ1—What are the most dangerous Classes of ATD in terms paid interest?

We provide a taxonomy (Figure 2) of the ATD classes that have been found, in practice, leading to costly interest in terms of sociotechnical phenomena and extra activities. The used research methodology assures that such map is based on recent and costly efforts experienced in practice by 9 organizations including several practitioners. The results are also supported by the analysis of 12 detailed cases of ATD.

RQ2—What kind of interest is triggered by different classes of ATD?

We provide a holistic model of the interest (Figure 2), and we show phenomena and activities generating extra costs triggered by the ATD classes. The model can be used as a quality model for developing context-specific methods or metrics on the basis of the triggered activities, to estimate the interest. The model provides a comprehensive representation of the interest considered by the practitioners when assessing the importance of ATD. The interest is a quite complex combination of several sociotechnical phenomena and extra activities. It is therefore important to report here such modeled complexity to help the conceptualization of interest by practitioners and researchers, as this has been a major challenge for the TD research community. Furthermore, the phenomena and activities reported in the model can be used as symptoms to recognize the presence of unknown ATD and as indicators to prioritize different ATD items generating different interest.

RQ3—Are there sociotechnical antipatterns that cause the ATD and its interest to increase over time causing the accumulation of compound interest?

By further analyzing the chains of events among causes of ATD, ATD classes, their effects and the resulting extra activities (Figure 2), we have discovered and modeled several patterns of events that cause a phenomenon that we have called contagious debt. Such sociotechnical patterns lead to the continuous increment of the ATD principal and interest: this causes the interest to be (potentially) compound. The impact of compound interest is substantial, as it might grow linearly or even exponentially, as reported in literature.[13] We have analyzed the spread of the principal and of the interest in 12 cases of contagious debt (reported in Table 2). This analysis has shown that the interest grows steeply and linearly in the cases and has also shown that such interest creates new ATD, which also grows steeply and linearly. This, by definition, leads to generate compound interest. In this study, we did not have the possibility to measure the resulting compound interest on the principal, as the growth was only estimated; the calculation of the compound interest would require the constant observation, in a longitudinal and more fine-grain study, of 3 main variables: principal, fixed interest, and compound interest (part of our future work). However, we report a holistic model that provides the targets that need to be observed (and possibly measured) to calculate the three variables, including the compound interest (see contributions for RQ4). Another contribution of this paper is the empirical evidence of the existence of sociotechnical patterns (contagious debt) leading to compound interest. In fact, in the cases reported here, we are able to explain how a theoretical and, so far, only financially formulated concept, such as compound interest, manifests itself in concrete software development cases. We can consider this as a very important milestone for TD research.

RQ4—How can contagious debt and, therefore, the accumulation of compound interest, be identified and stopped?

As a further contribution from the analysis related to RQ3, we report which software development factors cause the debt to be propagated and therefore to be contagious, potentially leading to compound interest. Such factors are the ones that need to be monitored by the practitioners to avoid the contagiousness and therefore the payment of high interest. This has a great effect on developing methods and tools, as shown by our current work.[12] The study has shown that using the propagation factors avoid contagious ATD to grow too much and therefore to accumulate compound interest.

5.3 | Further implications for research and industry

The results suggest a number of further practical and theoretical implications that are discussed below.

The results enable iterative and proactive ATD management. Constant and iterative monitor of ATD items, even if still not supported by powerful tools, is necessary for identifying the presence of ATD and for avoiding the payment of high interest. We propose here the taxonomy of ATD classes, interest phenomena and extra activities, contagious debt, and the propagation factors responsible for the increment of the interest. These results can be useful to support iterative architectural retrospective aimed at managing ATD. In such sessions, the models here serve as a guideline for practitioners who might recognize the presence of dangerous classes of ATD items. An example of successful method using such approach is reported in previous work,[12] which supports the importance of these results.

Prioritization of ATD items should be based on the contagiousness of the interest. There might be a lot of suboptimal solutions in a software system, and other studies show[22] that TD accumulation is not completely controllable and avoidable. The ATD classes, effects, and propagation factors proposed here help identifying and prioritizing ATD items that have more impact and therefore more interest to be paid. For example, the prioritization can be based on which ATD is going to accumulate compound interest and which are not. Such prioritization is important both for research and practice: as for the former one, our findings point at classes of ATD and interest in need for further research and for tool support (eg, new metrics), which are still quite scarce. As an example, we can see how many ATD items generate extra-testing activities: therefore, testing metrics should be used as a novel measure (combined with the known ones) to estimate the interest of an ATD item. As a practical effect, the development of metrics to assess the increasing and possibly compound interest will help practitioners in prioritizing the ATD with the highest estimated interest, which would save resources. This has an important implication: in fact, in software organizations, the available refactoring

resources are often largely fewer than the cost of repaying all the ATD present in the system. An impact analysis, using the propagation factors, can be particularly useful as input for the prioritization task performed by the management to understand when an ATD item has to be timely refactored instead of developing new customer-related features.

Predicting growth related to ATD is a key prevention. The main prevention for contagious debt and, in general, for increasing interest relies on understanding the growth of the propagation factors: the growth of the system (to understand the growing principal), the testware, the complexity, the users, and the system qualities connected to the injected ATD. This implication brings new knowledge in software engineering, because it expands the previously proposed theories on TD in which only maintainability is recognized as interest of an ATD item.[4] This study shows how ATD needs to be evaluated by calculating all the overall costs related to the interest: the focus on a single metric might result into a partial and skewed prioritization of ATD.

Technical debt in ecosystems might have greater impact. Some of the actors in the ecosystem need to assure the prevention of ATD. Another important consideration is the relation between the contagious debt phenomenon and the current evolving of big ecosystems, in which many different parties cooperate and compete. If we consider the ATD present in a single system as *epidemic* (borrowing the term from medicine, according to the *contagious* parallel used previously), the spread of the ATD in many systems can be considered as *pandemic*: in practice, ATD might result difficult to control if not contained in the beginning. We report three examples, in this study (Case_B_1, Case_B_2, and Case_B_3), in which ATD, present in part of the ecosystem (an organization using OS components), might spread to other organizations in the ecosystem (the OS community and other organizations using the OS component). For these reasons, we see the need, in future research, to identify architectural solutions (*quarantines*) that would avoid the spreading of contagious debt when the threat is identified. On the other hand, the presence of the open-source community participating in the ecosystems related to company B highlights how such development collaboration had a positive effect avoiding contagious ATD accumulation, thanks to the continuous external pressure of the OS community to focus on internal qualities of the software.

5.4 | Limitations

The results in this paper are the outcome of a large data collection effort prevalently using qualitative investigation, combined with a few quantitative results. Therefore, the results are not meant to substitute precise models derived from quantitative data, but rather to facilitate their creation. We did not aim, in this paper, at giving precise measurable results, but we contributed with a novel conceptualization of the interest of ATD, how high interest is generated and what leads to the accumulation of compound interest in software development. In software metrics, the creation of measurement systems and the collection of meaningful data need to follow a previously developed quality model. Our future work includes a deeper investigation of a limited number of cases to provide a more quantitative characterization of the phenomena. The taxonomy of ATD items might not be complete, because we focused on the most expensive violations mentioned by the practitioners. Also, different contexts might show different effects for the recognized items, in which case the taxonomy might result as different.

The possible threats to internal validity are the *temporal precedence*, *covariation*, and *nonspuriousness*. As for the first one, there is low possibility that the events described by the informants and checked with the architectural documentation would not be in the correct chronological order. As for the covariation, the results might have been affected by complex interrelation of variables, occurring in the cases, of which the researchers might not have been aware of. However, we have mitigated this problem by validating the models, comparing data coming from multiple companies, multiple sessions, and multiple cases across several sites. The same holds for the third threat, *nonspuriousness*, because it seems unlikely that the existence, in the studied cases, of other alternative explanations, especially when the phenomena were repeatedly mentioned across the sites. However, the threat is present. As for external validity, although we cannot generalize the results, we can rely on a higher number of organizations (9) have been studied, which is higher than in many other works in literature: in most studies, usually a single or few case studies are taken in consideration. The last threat is the *confirmation bias*, which might have occurred during the validation step: when the researchers proposed the models, the respondents might have tried to fit examples with the wish of "believing" in the proposed model. This threat has been mitigated by constant triangulation from several sources and when studying the in-depth cases reported in Table 2 and their fit with the definition of Contagious Debt.

6 | RELATED WORK

The phenomenon of ATD has recently received the attention of the research community.[2,3] It was difficult, therefore, to find extensive previous research tackling the problem of prioritizing ATD items on the basis of their interest, especially related to real contexts. Few single case studies were related to code debt (eg, Guo et al[24]) or code smells.[25,26] The work done by Sjøberg et al shows that some code smells do not create extra maintenance effort: the implications are that not all the "smells" or architecture quality issues, identified in literature, leads to a necessarily high extra effort: we followed such idea on an architecture level, and we empirically classified ATD creating more extra effort (for prioritization purposes). Some prioritization work has been done in previous work,[27] where the authors ask the developers to prioritize quality rules derived from a static analysis tool, which however covers source code debt. Our study is similar in the ATD domain, as we asked the practitioners to give an in-depth estimation of the interest. In the case study conducted by Nord et al,[7] the authors studied a single and detailed case of ATD, modeling and developing a metric on the basis of architectural rework that would help deciding between different paths leading to different outcomes. This initial attempt to create a metric focuses on the impact of modularity on

the interest to be repaid in the future. Although the resulting metric seems promising, the scope of such paper is specific for one class of ATD (ATD3 in our taxonomy) and does not attempt to classify different ATD items on the basis of their effects. Furthermore, it does not take into account compound interest and what leads to it. Another study that attempts to calculate the interest on the same kind of debt is the work of Kazman et al.[9] However, although the study is able to quantify ATD belonging to class ATD3, the calculated interest is partial, because only the interest related to bug fixing is taken in consideration (effect E7 in our taxonomy). Finally, the study by Arvanitou et al[28] introduces a ripple effect measure; such measure can be useful to quantify the estimated effect of a change and can therefore be useful when measuring spread changes, E5 in our taxonomy; it can be used to estimate the probability of changes related to another change. Calculating the changes can also reduce unawareness of the hidden ripple effects (P8) and can help in estimating what needs to be refactored. However, it is important to notice that such measure does not help in estimating the upcoming interest. Many factors need to be considered when estimating the interest, and some of them cannot be retrieved from source code artifacts (for example, the amount of future features to be developed). Most of the previous work is based on analyzing the source code, but none of these studies attempt to understand the sociotechnical phenomena that lead to the accumulation of ATD and compound interest. On compound interest, only a code debt paper was previously published, which takes in consideration compound interest as the growth of the system (but it does not mention another propagation factors explained here). Compound interest is introduced in previous work[13] as a theoretical concept. However, as highlighted in literature[4], there is a need for more empirical studies on the subject; our paper can be considered the first in identifying empirical evidence of the existence of compound interest in practical industrial cases and to show which sociotechnical phenomena can potentially lead to it. The studied cases also tell us that it is quite important to increase the holistic understanding of the accumulation of interest to create concrete measures or estimation methods to support the architects, but unfortunately, we could not find any study on the subject.

We find, as an interesting point of discussion, the difference between the intentional and unintentional debt, as introduced by Fowler,[29] and the assumption, in more than one work (eg, Nord et al[7]; Nugroho et al[30]) that unintentional debt is linked to code debt while architectural debt is necessary strategically chosen and intentional. However, our empirical findings, together with a previous study,[22] suggest that also at the architectural level there is an accumulation of unknown ATD (obviously unintentional). For example, members of the teams are not always aware of the impact of low-level choices to the whole system architecture. An empirical model of debt and interest is also described in previous work,[30] which focuses on a generic quality level on the basis of the maintenance effort. Such model, although useful as inception for our empirical investigation, results quite simplistic, because it does not contain the same information that can be found in our models, and it does not provide a classification and prioritization of different ATD items. Furthermore, it does not take in consideration the organizational- and processes-related phenomena connected to the accumulation of ATD interest. Our work has been inspired by the work done by Seaman et al,[10] Gou and Seaman,[20] and Zazworka et al[21]. In the work of Seaman et al[10] a model of cost/benefits has been proposed for ranking ATD items. Our results suggest that the proposed models would need to include the time dimension, as we have found that the interest might grow in different ways, and the inclusion of specific properties such as the "contagiousness" of the ATD item. In summary, none of the studies found in literature took a holistic perspective as we have done in this paper, taking in consideration a broad landscape of sociotechnical phenomena.

7 | CONCLUSIONS

Strategic decisions on the prioritization of costly ATD are critical for software companies. To prioritize the refactoring of ATD, companies need to be aware of what *interest* is being paid and what interest is going to be paid in the future. The current body of knowledge lacks a solid conceptual representation of the interest on the basis of empirical studies, and the community lacks applicable approaches to estimate the interest and to prioritize ATD: as a result, the architects' decisions are based on individual experience and intuitive judgment, and the repayment of ATD is consequently hard to justify for the management. Estimating the extra costs associated with the interest requires knowledge about what phenomena and extra activities are generated by the ATD. In this study, we have collected a broad amount of cases where large software companies experienced on the payment of large amount of interest. We have categorized the ATD in classes and mapped them to costly interest represented by sociotechnical phenomena occurring during software development and extra activities generating costs. The paper reports patterns of sociotechnical events that lead to a novel phenomenon, call *contagious debt*; this is based on vicious cycles of events that cause ATD to propagate in the system and generate a continuous increment of interest. We report evidence that this phenomenon is a potential trigger for the accumulation of *compound interest*, the source of fast-growing hidden costs, which might become nonlinear over time. This paper can be considered the first study in reporting a large amount of empirical evidence on the occurrence of compound interest and of the sociotechnical patterns leading to it. Our results provide clear targets for which new information and, consequently, a new combination of measures are needed to estimate the interest and the compound interest of ATD. Such results can be valuable in making practitioners aware of what kind of interest they might have to pay if the ATD is not repaid. Finally, we provide a list of the factors that cause the debt to propagate throughout the system, becoming contagious. These factors can be proactively monitored to strategically refactor the ATD, before it becomes too costly to be repaid. Our recent work, involving methods and tools for the estimation of the interest on the basis of the factors outlined here, reported several benefits of using such information during software development.

ACKNOWLEDGMENTS

We thank the companies that are partners of the software center and especially the architects group involved in the project run by the

researchers and authors of this article. We would also thank Mattias Tichy, Michel Chaudron, and Lars Pareto for their suggestions and review of the research process.

REFERENCES

1. Cunningham W. The WyCash portfolio management system. In: *ACM SIGPLAN OOPS Messenger*. 1992;4:29-30.
2. Tom E, Aurum A, Vidgen R. An exploration of technical debt. *J Syst Softw*. 2013;86(6):1498-1516.
3. Kruchten P, Nord RL, Ozkaya I. Technical debt: from metaphor to theory and practice. *IEEE Softw*. 2012;29(6):18-21.
4. Li Z, Avgeriou P, Liang P. A systematic mapping study on technical debt and its management. *J Syst Softw*. 2015;101:193-220.
5. Besker T, Martini A, Bosch J. A systematic literature review and a unified model of ATD. In: *2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Limassol, Cyprus; 2016:189-197. https://doi.org/10.1109/SEAA.2016.42
6. Avgeriou P, Kruchten P, Nord RL, Ozkaya I, Seaman C. Reducing friction in software development. *IEEE Softw*. 2016;33(1):66-73.
7. Nord RL, Ozkaya I, Kruchten P, Gonzalez-Rojas M. In search of a metric for managing architectural technical debt. In: *2012 Joint Working IEEE/IFIP Conference on Software Architecture (WICSA) and European Conference on Software Architecture (ECSA)*, Helsinki, Finland; 2012:91-100.
8. Martini A, Bosch J, Chaudron M. Investigating architectural technical debt accumulation and refactoring over time: A multiple-case study. *Inf Softw Technol*. 2015;67:237-253.
9. Kazman R et al. A case study in locating the architectural roots of technical debt. In: *Proceedings of the 37th International Conference on Software Engineering-Volume 2*. NJ, USA: Piscataway; 2015:179-188.
10. Seaman C et al. Using technical debt data in decision making: potential decision approaches. In: *2012 Third International Workshop on Managing Technical Debt (MTD)*, Zurich, Switzerland; 2012:45-48.
11. Schmid K. A formal approach to technical debt decision making. In: *Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures*, Vancouver, Canada; 2013:153-162.
12. 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 Companion*, Austin, USA; 2016:31-40.
13. Ampatzoglou A, Ampatzoglou A, Chatzigeorgiou A, Avgeriou P. The financial aspect of managing technical debt: a systematic literature review. *Inf Softw Technol*. 2015;64:52-73.
14. Runeson P, Höst M. Guidelines for conducting and reporting case study research in software engineering. *Empir Softw Eng*. 2008;14(2):131-164.
15. Dubois A, Gadde L-E. Systematic combining: an abductive approach to case research. *J Bus Res*. 2002;55(7):553-560.
16. Yin RK. *Case Study Research: Design and Methods*. Los Angeles, London, New Delhi, Singapore, Washington DC: SAGE; 2009.
17. Martini A, Sikander E, Madlani N. Estimating and quantifying the benefits of refactoring to improve a component modularity: a case study. presented at the SEAA Euromicro 2016; 2016.
18. Strauss A, Corbin JM. *Grounded Theory in Practice*. London, New Delhi: SAGE; 1997.
19. Martini A, Bosch J. The danger of architectural technical debt: Contagious debt and vicious circles. In *accepted for publication at WICSA*. Montreal, Canada; 2015.
20. Guo Y, Seaman C. A portfolio approach to technical debt management. In: *Proceedings of the 2Nd Workshop on Managing Technical Debt*. New York: NY, USA; 2011:31-34.
21. Zazworka N, Spínola RO, Vetro' A, Shull F, Seaman C. A case study on effectively identifying technical debt. In *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*, New York, NY, USA; 2013:42-47.
22. Martini A, Bosch J, Chaudron M. Architecture technical debt: understanding causes and a qualitative model. In: *40th Euromicro Conference on Software Engineering and Advanced Applications*. Verona; 2014:85-92.
23. Mamun MAA, Berger C, Hansson J. Explicating, understanding and managing technical debt from self-driving miniature car projects. In *Proceedings of Sixth International Workshop on Managing Technical Debt*. Victoria, British Columbia, Canada; 2014.
24. Guo Y et al. Tracking technical debt—an exploratory case study. In: *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, Williamsburg, VI, USA; 2011:528-531.
25. Sjoberg DIK, Yamashita A, Anda BCD, Mockus A, Dyba T. Quantifying the effect of code smells on maintenance effort. *IEEE Trans Softw Eng*. 2013;39(8):1144-1156.
26. Arcelli FF, Dietrich J, Walter B, Yamashita A, Zanoni M. Preliminary catalogue of anti-pattern and code smell false positives. Pozna'n University of Technology., Technical Report RA-5/15; 2015.
27. Falessi D, Voegele A. Validating and prioritizing quality rules for managing technical debt: an industrial case study. In *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*; 2015:41-48.
28. Arvanitou EM, Ampatzoglou A, Chatzigeorgiou A, Avgeriou P. Introducing a ripple effect measure: a theoretical and empirical validation. In: *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, Beijing, China; 2015:1-10.
29. Fowler M. Technical Debt Quadrant; 2009.
30. Nugroho A, Visser J, Kuipers T. An empirical model of technical debt and interest. In *Proceedings of the 2nd Workshop on Managing Technical Debt*. New York, NY, USA; 2011:1-8.

How to cite this article: Martini A, Bosch J. On the interest of architectural technical debt: Uncovering the contagious debt phenomenon. *J Softw Evol Proc*. 2017;e1877. https://doi.org/10.1002/smr.1877