# The Danger of Architectural Technical Debt: Contagious Debt and Vicious Circles

Antonio Martini
Computer Science and Engineering, Software Engineering
Chalmers University of Technology
Göteborg, Sweden
antonio.martini@chalmers.se

Jan Bosch
Computer Science and Engineering, Software Engineering
Chalmers University of Technology
Göteborg, Sweden
jan.bosch@chalmers.se

*Abstract* — A known problem in large software companies is to balance the prioritization of short-term with long-term viability. Specifically, architecture violations (Architecture Technical Debt) taken to deliver fast might hinder future feature development. However, some technical debt requires more interest to be paid than other. We have investigated which Technical Debt items generate more effort and how this effort is manifested during software development. We conducted a multiple-case embedded case study comprehending 7 sites at 5 large international software companies. We found that some Technical Debt items are contagious, causing other parts of the system to be contaminated with the same problem, which may lead to non-linear growth of interest. We also identify another socio-technical phenomenon, for which a combination of weak awareness of debt, time pressure and refactoring creates Vicious Circles of events during the development. Such phenomena need to be identified and stopped before the development is led to a crisis point. Finally, this paper presents 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.

*Index terms*—architectural technical debt, agile software development, effort, socio-technical phenomena, multiple case-study, qualitative model.

## I. 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. The trend in the last decade has been the employment of Agile Software Development (ASD) [1]. 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 trend of taking sub-optimal decisions in order to meet short-term goals to the taking debt, which has to be repaid with interests in the long term. Such a concept is referred as Technical Debt (TD), and recently it has been recognized as a useful basis for the development of theoretical and practical frameworks [2]. Tom et al. [3] have explored the TD metaphor and outlined a first framework in 2013. Part of the overall TD is to be related to architecture sub-optimal decisions, and it's regarded as Architecture Technical Debt (ADT)[2]. ATD is regarded as violations in the code towards the intended architecture for supporting the business goals of the organization. An example of ATD might be the presence of structural violations [4].

ATD has been recognized as part of TD, but the various violations have not been compared among each other in literature with respect to costs and time. A dedicated study about which ATD items are the most dangerous, in terms of interests to be paid, is still missing. In fact, given the expensiveness of architectural changes, a challenge for software companies is to prioritize the ones that are really needed, in order to optimize the employment of resources. Moreover, it's important to understand what kind of interest (in terms of effort) is associated with the ATD items, both for their recognition during development and for the development of measurements.

In the context of large-scale ASD, the research questions that we want to inform are:

RQ1: What are the most dangerous Architecture Technical Debt Items (violations) in terms of effort paid later?

RQ2: What are the effects triggered by such ATD items?

RQ3: Are there socio-technical patterns of events that trigger the creation of ATD leading to particularly dangerous interest to be paid?

In this paper we have employed a 18-months multiple-case embedded case-study involving 7 different Scandinavian sites in 5 large international companies in order to shed light on which Architectural Technical Debt items are the most dangerous (in terms of effort) and how such effort is manifested during software development. We have analyzed the qualitative data coming from more than 30 hours of

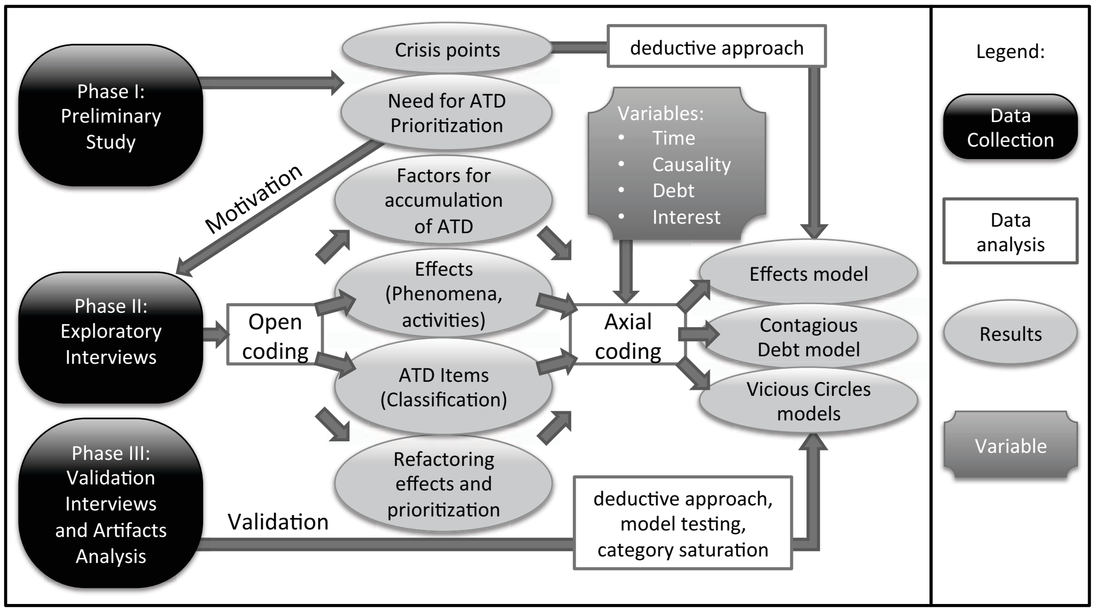

Phase I: Preliminary Study
Crisis points
deductive approach
Need for ATD Prioritization
Legend:
Data Collection
Variables:
• Time
• Causality
• Debt
• Interest
Motivation
Factors for accumulation of ATD
Data analysis
Effects model
Phase II: Exploratory Interviews
Open coding
Effects (Phenomena, activities)
Axial coding
Contagious Debt model
Results
ATD Items (Classification)
Vicious Circles models
Variable
Phase III: Validation Interviews and Artifacts Analysis
Refactoring effects and prioritization
deductive approach, model testing, category saturation
Validation

**Figure 1.** Our Research Design: data collection, inductive and deductive analysis and results based on different sources for triangulation

interviews complemented by existing architecture documentation, using a combination of inductive and deductive approach proper of Grounded Theory.

The main contribution of the paper is threefold: we have qualitatively developed and validated (through multiple sources) a taxonomy of effortful ATD items to inform RQ1. We model the effects associated to the classes in the taxonomy, to inform RQ2. To inform RQ3, we have found and conceptualized two important phenomena of Architectural Technical Debt, contagious debt and vicious circles, which are related to the occurrence, during the development, of dangerous patterns of socio-technical events that may lead to the non-linear growth of interest to be paid over time.

## II. Research Design

We conducted a 18-monhts long, multiple-case, embedded case study involving 7 Scandinavian sites in 5 large international software development companies. We decided to collect data from as many large companies as we could, in order to increase the degree of source triangulation [5] (collecting supporting evidence from different sources rather than from a single context). The nature of the study was exploratory, since the lack of previous literature focusing on the specific research problem. Therefore, we wanted to maximize the coverage of possible software companies within the boundaries of "large" and "Agile", in order to capture as many ATD items experienced by the companies, but at the same time having the opportunity to dig out as many details from the context as possible. The research design is outlined in Figure 1.

### A. Case Selection

We have employed an embedded multiple-case study [5], where the unit of analysis is an (sub-part of the) organization: the unit needed to be large enough, developing 2 or more sub-systems involving at least 10 development teams. The total units studied were 7. We selected, following a literal replication approach [6], 4 companies: A, B, C (3 sub-cases) and D, large organizations developing software product lines, having adopted ASD and had extensive in-house embedded software development. We also selected company E, a "pure-software" development company, for theoretical replication [6] (hypothesizing different results from the other companies).

*Company A* is involved in the automotive industry. The development in the studied department is mostly in-house, recently moved to SCRUM. *Company B* is a manufacturer of recording devices. The company employed 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 3 different departments within company C ($C_1$, $C_2$, $C_3$). *Company D* employed 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 employed SCRUM.

### B. Data collection

We planned a 3-phase investigation of the ATD items and effects. The three phases (black boxes) and their results are visible in Figure 1. The properties and the number of participants are also summarized in Table 1.

Phase I – We conducted a preliminary study involving 3 of the abovementioned cases, in particular A, $C_1$, and $C_2$, in which we explored the needs and challenges of developing and maintaining architecture in an Agile environment in the current companies. We organized three 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

TABLE I. Properties and numbers related to data collection

| Phases of data collection | N. participants | Hours of data | N. sessions | Companies | Roles involved |
|---|---|---|---|---|---|
| Phase I (Preliminary interviews) | 25 | 12 | 3 | Company-specific | Developers, architects, testers, line managers, Scrum m. |
| Phase I (Validation workshop) | 40 | 4 | 1 | Cross-company | Developers, architects, line managers |
| Phase II (group interviews) | 26 | 14 | 7 | Company-specific | Developers, architects, product owners |
| Phase II (validation workshop) | 10 | 3 | 2 | Cross-company | Architects, line managers |
| Phase III (validation interviews 1) | 10 | 4 | 1 | Cross-company | Architects, product owners |
| Phase III (validation interviews 2) | 12 | 4 | 1 | Cross-company | Architects, developers, scrum masters |
| Phase III (validation workshop) | 20 | 4 | 2 | Cross-company | Architects |
| Informal interaction | 7 | NA | NA | Company-specific | Software and system architects |

validated and discussed in a final one-day workshop involving 40 representatives from all the 7 cases (see Table 1).

In the preliminary study we asked open questions such as "How do you control consistency between the implementation and the architecture?", "Which architecture risk management activities are you employing on different level of abstraction?" and "How do you prioritize architecture improvements?". This set of question aimed at understanding what architecture practices were currently employed in the organizations in order to identify the architecture debt (consistency), its interest (risk of effort) and how it was prioritized. The preliminary study showed 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.

Phase II – In the second phase we conducted 7 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, in order to cover many aspects: the source of ATD (developers), the architectural implications (architects and system engineers), the prioritization decisions taken (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 architecture inconsistencies (ATD items) with high effort impact. We took a retrospective approach: we aimed at identifying real cases happened in the recent past rather than rely on speculations about the future. We asked, in order, "Can you describe a recent major refactoring, a high effort perceived during feature development or during maintenance work?", "Does such effort lead to architecture inconsistencies?" and "What are the root causes for the identified architecture inconsistency?". The output was a list of ATD items with large effort impact. Then, for each identified architecture debt item, we followed-up with the developers involved in the development in order to understand the in-depth details.

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. We have found no other studies applying such technique, which adds methodological novelty to our paper.

Phase III – The third phase consisted of two validation activities: we organized 3 multiple-company group interviews, including all the roles involved in the investigation, developers, architects and product owners, where we showed the models for their recognition and improvement. For example, we proposed the model for contagious debt (section III.B.1)) and we asked, when recognized, to strengthen the model with further concrete and real examples. In order to test the completeness of the data, we also included, where possible, the analysis of artifacts such as lists of *Technical Issues* or *Architectural Improvement* identified within the company, in order to understand if the identified items were mapped to the developed taxonomy. Such deductive procedure strengthened the inductive process employed in the first and second phases, leading to category saturation, an important prerequisite for the development of grounded theories.

As a further validation step we organized 2 plenary workshops with around 20 architects also from 2 other large companies not previously participating in the study, in order 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 support cases to validate the models.

## C. Data analysis

The workshops were recorded and transcribed. The analysis was done following an approach based on Grounded Theory [7] and using a tool for qualitative analysis, to keep track of the links between the codes and the quotations they were grounded to. Given the exploratory nature of our study and our need to develop novel concepts and theories about ATD, we opted for the use of the following methods, frequently employed for the analysis of large amount of semi-structured, qualitative data representing complex combinations of technical and social factors.

Open Coding – First we analyzed the data in search for emergent concepts following open coding, which would bring novel insights on the analyzed issue. We coded the identified ATD items in the taxonomy. We used the same technique for identifying the key effect phenomena and the causing factors that were related to each item. The second inductive step comprehended the grouping of codes into more abstract or special categories, for example the classes of the taxonomy of ATD items. Such analysis led to the taxonomy, the effects phenomena and activities (Figure 2) and the elements in the modeled *vicious circles*.

Axial Coding – The codes and categories were compared through axial coding in order to highlight connections orthogonal to the previous developed categories. Such analysis showed which category of items (in the taxonomy) was connected to which effects. This way we could build the complete effects model in Figure 2, in which we could represent the *debt* and the *interest* discovered through our investigation. This step was also used for modeling the links between events and causing factors. The results were extremely precious in recognizing the presence of patterns such as the *Contagious Debt* and the presence of *Vicious Circles* (better explained in the results part) that showed how ATD could lead to the payment of non-linear interest.

Deductive Approach – Some of the companies (three of them) maintained documents, containing the informal description of the architectural issues (inconsistencies) currently recognized. Such items were mapped to our taxonomy in order to check its completeness. This process strengthened the previous inductive analysis, also providing triangulation of resources. We applied the deductive approach also for testing the models of *Contagious Debt* and the *Vicious Circles*: we used several concrete instances (not reported completely here for space reasons) from the investigation to test if they were fitting the models. An example of such step can be found in section III.B.1), where a concrete case is used for the application of the *Contagious Debt* abstract model.

## III. Taxonomy of Architecture Technical Debt Items, their Effects and Vicious Circles in The Model

The ATD accumulated in the system is commonly represented by *Items*, also according to recent studies on Technical Debt management [8],[9]. We discovered that some kinds of items were connected to the occurrence of certain key *phenomena* in software development. The results also show connections between such phenomena and the effects in terms of triggered development *activities*. The overall model (Figure 2) helps the visualization of such relationships. The paths in the model represent the connection between the *debt* and the *interest*, relationship that is of utmost importance for prioritizing the items to be refactored. Through such model we will also be able to show the real danger of some ATD items: the phenomenon called *contagious debt* and the *vicious circles* highlighted by the links and loops between the various elements of the model. This will show dangerous accumulation of ATD interest (potentially non-linear, Figure 3), which can be considered a major threat for software development.

### A. *Taxonomy of ATD Items and their effects*

The ATD items identified during the investigation can be grouped in the following categories. We do not list all the gathered items for space reasons, but for each category we give a short explanation and a representative example. We also link the classes of items with the triggered phenomena and activities. The overall model is shown in Figure 2.

#### *1) Dependency violations and unawareness*

This category includes the items that are related to the presence of architectural dependencies (for example at different component levels) which are considered forbidden in the (context-specific) architecture. An example of this class of items is represented by a component that, when executed, should not trigger the execution of another component, as specified by the architects/architecture. Examples of these case were mentioned by all the interviewed companies.

In case $C_2$ the interviewees mentioned a concrete example of the phenomenon connected with this kinds of items: the large amount of dependencies among the components in one of the sub-systems caused, each time a new release involved a small change, the test of the whole sub-system (*Big deliveries* 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).

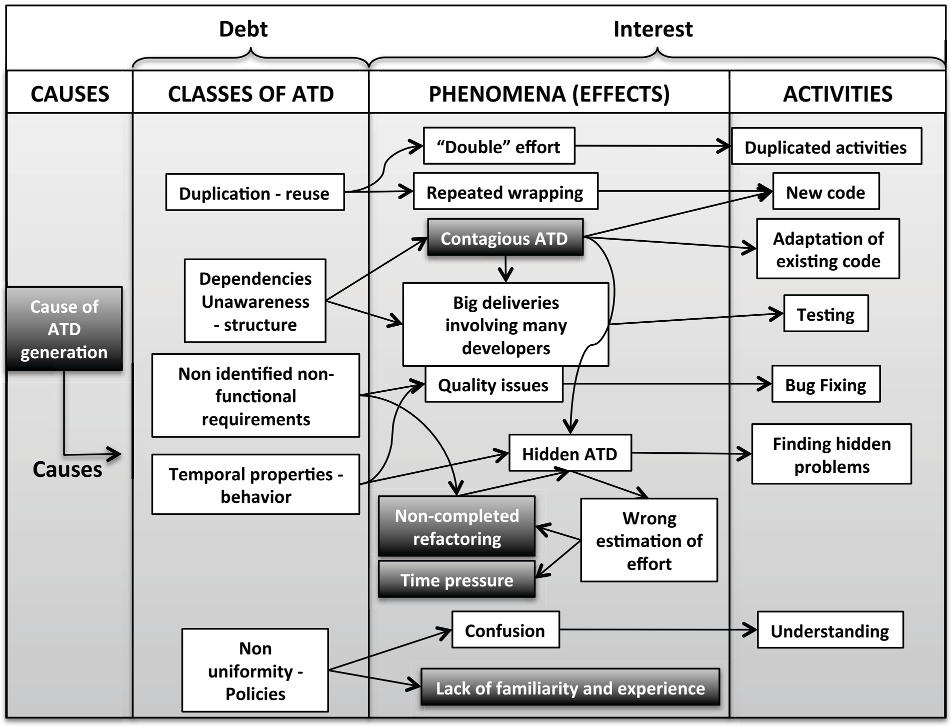
This category also includes those items that are connected to other parts of the system through the presence of dependencies that are not reconginzed by the developers and architects, and therefore cannot be correctly located in a specific part of the code. The main problematic effect related to this issue is that the actual ATD item spreads out in the system as the system grows, making both the cost of removing it and of its effects growing constantly: for this reason, we call it *contagious debt*. Moreover, it creates *hidden* ripple effects that are caused by the chains of interactions that the discovered ATD item is connected to. Since this phenomenon also represents one of the more dangerous vicious circles that we have found, a concrete example is analyzed in section III.B.1).

The developers and architects, although aware of the ATD item, are usually not aware of the degree of "contagiousness" of such item, which would make it hidden (*hidden ATD* in the model). In some cases, the ATD item present in the original component would also trigger the creation of additional code in order to adapt 3d party components (for example open source or supplied software).

#### *2) Non-uniformity of patterns and policies*

This category comprehends patterns and policies that are not kept consistent through 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 the same functionality. As a concrete example we bring Case A, where different components (ECUs in the automotive domain) communicated through different patterns.

The effects caused by the presence of non-uniform policies and patterns are of two kinds: the time spent by the developers



Debt
Interest
CAUSES
CLASSES OF ATD
PHENOMENA (EFFECTS)
ACTIVITIES
Cause of ATD generation
Causes
Duplication - reuse
Dependencies Unawareness - structure
Non identified non-functional requirements
Temporal properties - behavior
Non uniformity - Policies
"Double" effort
Repeated wrapping
Contagious ATD
Big deliveries involving many developers
Quality issues
Hidden ATD
Non-completed refactoring
Wrong estimation of effort
Time pressure
Confusion
Lack of familiarity and experience
Duplicated activities
New code
Adaptation of existing code
Testing
Bug Fixing
Finding hidden problems
Understanding

**Figure 2**. The model shows the causes for ATD accumulation (black boxes), the classes of ATD (which represent the Debt), the phenomena caused by the items and the final activities (which together represent the interest to be paid).

in understanding parts of the system that are not familiar with and by understanding which pattern to use in similar situations. For example, in case A, the developers experienced difficulties in choosing a pattern when implementing new communication links among the components, since they had different examples in the code.

A phenonmenon involved specifically the feature teams interviewed at company C. In such context, the teams were completely unlinked from the architectural structure (each team could "touch" 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. This phenomenon also leads to a vicious circle, as explained in section B.

*3) Code duplication (non-reuse)*

Some ATD items were related to 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 leading to double maintenance.

Another important phenomenon related to code duplication is the presence of what has been called, during the interviews (citing an architect from company B), "glue code". Such code is needed to adapt the reused component to the new context with new requirements. Such code is usually unstructured and sub-optimal, because not part of the architectural design but rather developed as a workaround to exploit reuse: however, the risk is that, with the continuous evolution of the system around the reused component, such glue code would evolve uncontrolled, becoming a substantial part of the (sub-)system, which would contain ATD.

At the same time, some of the developers from different companies mentioned the fact that the lack of reuse (citing interviewees, "copy-paste") is not always a bad solution. As mentioned earlier, extensive reuse might lead to the presence of glue code. It might be more desirable to reuse the code to save development time but evolve it without keeping it dependent to the original component.

*4) Temporal properties of inter-dependent 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: it creates a number of quality issues 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). However, the worse danger of this problem is related to the temporal nature of it. Being a behavioral issue, 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 understanding that it's the source of the problems (explicitly mentioned by company 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 sub-optimal solution. The *hidden* nature of these ATD items create a number of other connected effects (as explained in III.B.2)).

*5) Unidentified non-functional requirement*

Some non functional requirements, such as performance and signal reliability, need to be recognized before or 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 case C mentioned the lack of a fault handling mechanism, which was very expensive to be added afterwards, together with lack of testability mechanisms. Case A reported frequent struggles with non functional requirements regarding memory consumption and communication bus allocation. Case B mentioned the difficulties in anticipating the future use cases for a given feature.

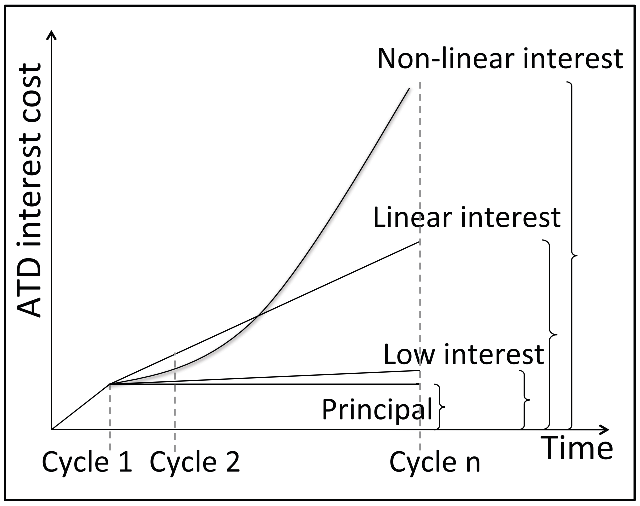


**Figure 3**. Different increments of interest over time with respect to several cycles of Vicious Circles: principal (fixed), low interest, linear interest and non-linear interest.

The introduction of such debt causes a number of quality issues, 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: quantifying the change and estimating the cost of refactoring has always been reported as a challenge, which brings to other problems as explained in the next section.

*B. Vicious Circles*

In section A we have described the connection of 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 association can be considered as *fixed*, which means that each item brings a constant cost. However, our analysis of the relationships among the different phenomena have brought to light patterns of events that are particularly dangerous, because they create loops of causes-effects that lead to linear and potentially non-linear accumulation of interest, which might virtually have no end or, more probably, might result into a crisis [10]: they are *vicious circles*.

Such vicious circles are well visible in our model in Figure 2: the column on the left includes the possible causes of ATD accumulation ([10] for details), which we have represented with black boxes (*Cause of ATD generation*). Among the phenomona triggered by some classes of ATD, we can find also causes of ATD (also represented by black boxes). This means that when a path starts with a cause (all ATD items have a cause), pass through some ATD items and ends in a phenomenon that is also a black box, such black box also causes the creation of additional ATD. This loop represents the vicious circle.

The implications of vicious circles are important, since the presence of vicious circles implies the constant increment of the ATD items and therefore of their effects over time. Which means, each moment that passes with the ATD items involved in the vicious circles remaining in the system, the interest increases. Such phenomenon causes the ATD to become very expensive to be removed afterwards, together with the hindering effects over the development speed: as shown in Figure 3 for each cycle of the vicious circle, the cost might remain constant (for example the *principal* of fixing the ATD item), linear but with low increment over time (*low interest*), linear but with a high steepness (*linear interest*) or it might even reach non-linearity (*non-linear interest*). In the end, such accumulation might bring to a crisis.

The highlighted vicious circles are:

*1) Contagious ATD*

We have introduced the concept of contagious ATD in section III.A.1). In order to show the danger of contagious ATD, we first describe an example from real world (chosen among others), then we derive a model and simulate an instance with fictious values. Even though intended for illustrative purposes, such projection clearly shows the potential danger related to the interest accumulated through this phenomenon.

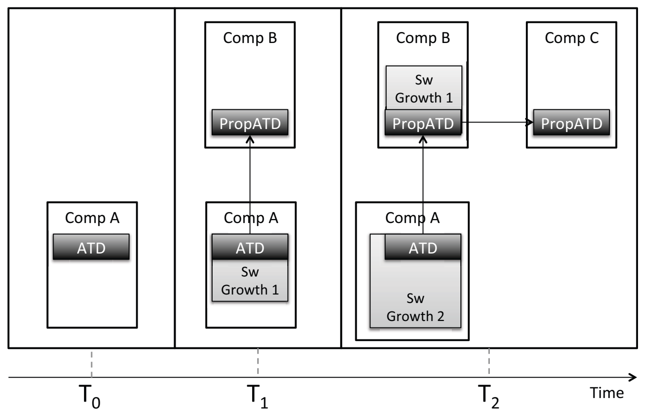


**Figure 4**. Model for Contagious Debt accumulation at different points in time: $T_1$, $T_2$ and $T_3$

Although we have found many concrete examples during our investigation, we have chosen the one that seemed more representative for this phenomenon. In this case, according to the case-specific desired architecture, a database in a layered architecture (we will refer to it as "component A") could 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, the database component was created without a standardized interface, which was the actual ATD item. In the system, there was another component accessing the database A (we will call it "B"), which would use the non-standard, direct interface provided by A everytime B wanted to interact with 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 obviously 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 had to interact with the database (comp 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 at this point in time, would involve the refactoring not only of the database (component A containing the ATD) but also of component B, interacting with it: clearly, the interest was much higher than changing only A. On the other hand, not removing the ATD item, would have meant spread it to component C, D etc. making its removal even more costly. The interest, when the company would have decided to change the database according to its roadmap, would have raised much more, since the removal of ATD from A would have meant the removal of it also from B, C, D, etc.

From the concrete cases, we built the model showed in picture Figure 4: 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 and called, according to the financial metaphor, 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). At this point in time, 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 three 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 fictious 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's show what happens if we apply this model to the previous concrete case: at $T_0$, when the component was created with 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 the components grow constantly, making it a worse case scenario, but we have to consider the fact that also more components could have been added.

Comparing this case with the cost model in Figure 3, it suggests a quite steep growth of the interest increment, which might even become non-linear in the worst case. It becomes of utmost importnance, 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.

*2) Hidden ATD, not Completed Refactoring and Time Pressure*

Hidden ATD is a common cause of more than one vicious circle: what happens is that some ATD items cause the creation of hidden ATD (for example, the ripple effects created by contagious debt). Then the unawareness of the hidden debt, by the developers and architects, causes them to be unable to estimate correctly the time to refactor or to deal with the ATD item. Consequently, when a refactoring is planned for a certain period, it might result in being incomplete or when new features are added where ATD is located, the time for delivering such features might increase. Incomplete refactoring has been recognized to be a cause for new ATD accumulation in [10] and [11] (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 have picked a concrete example of this phenomenon described by the informants at one studied site: the architects identified an ATD item consisting in a violation for which three different patterns were used to communicate among components in different parts of the system. Such problem had shown to create difficulties during development, since developers felt confused about when using one or another. Therefore, a refactoring was planned by the architects in order to remove the three different protocols and replace them with a unique fourth one, consistent all over the system. However, during the refactoring, several ripple effects where discovered that were connected to the implementation of the three patterns to be removed. Such effects were not considered during estimation time. Given the time pressure to finish the refactoring, the result was having a fourth protocol included in the system without the developers being able to remove the other three. Even worse was the fact that the management would not prioritize such refactoring again, given that the problem was meant to be solved.

This example clearly shows how the presence of hidden ATD would lead to the inclusion of even more ATD in the system, making it a vicious circle and creating a trend of continuous increment of ATD and interest to be paid.

#### 3) *Lack of uniformity and familiarity*

As highlighted in the model, 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 more accumulation of ATD. Although this chain of events might not create continuous growth of interest, it's worth highlighting that this particular combination *might* lead to a vicious circle in the worst cases. However, it also shows how ATD and its interest, in such situation, might increase more than it is perceived intuitively.

## IV. Discussion

The exploratory investigation that we have carried out contributes to inform the research questions with the following results: by providing a taxonomy of the architectural violations (ATD) that have been found in practice to lead to phenomena and activities connected to high effort cost (RQ1). The employed research methodology assures that the items come from recent costly efforts experienced in practice by 7 organizations. To inform RQ2 we have built a model of the effects (Figure 2) in which we show the phenomena connected to the items and the activities that are triggered. The model can be used as a quality model for developing context-specific metrics and other artifacts in the companies in order to estimate and therefore to prioritize the ATD items based on what they may cause in the future. To inform RQ3, we have further analyzed the relationships, discovering and modeling pattern of events such as the contagious debt and vicious circles that cause the continuous increment of ATD interest to be paid over time, which might be linear but could potentially reach non-linearity, leading to development crisis.

### A. *Implications for research and industry*

The results suggest a number of practical and theoretical implications, which might evolve, in future work, into practices for practitioners and new research studies. First of all, 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 dangerous phenomena described here. Architectural retrospective, led by architects, should be run every number of iterations, according to the abstraction level of the ATD items to be checked for. In such sessions, the models here serve as a guideline for practitioners who might recognize the presence of dangerous classes of ATD items or the verifying of the phenomena (that might be interpreted as a symptom or as a warning). Especially important is the recognition of the presence of hidden ATD, for example by checking the unusual difference between the estimation and the actual time employed for carrying out a refactoring or delivering a feature. Very dangerous, according to our model of vicious circle, is leaving a refactoring incomplete. Another important phenomenon is the recognition, through the architectural retrospective, of the "contagiousness" of the identified ATD items. In such dangerous cases, not only the teams and the architects should be aware of the danger, but the information needs to be signaled to managers, who need to be aware of the situation and take decisions about budget allocation for refactoring according to the severity of the revealed phenomena.

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. The risk that the ATD present in a single system (which would be *epidemic*, borrowing the term from medicine) would spread in many systems in a *pandemic* fashion might result in a phenomenon difficult to control once not contained in the beginning. For this reasons we see the need, in future research, to identify architectural solutions that would avoid the phenomenon, but also stop the spreading once the contagious ATD item would be identified.

There is another point that we find worth mentioning: the graphs in picture Figure 3 show different trends of interest increment over time. However, such functions are related to the effects of *single* items. According to another study [10], there might be different causes of ATD accumulation. Such causes are: *Uncertainty of use cases in the beginning, Business evolution creates ATD, Time pressure: deadlines with penalties, Priority of features over product, plit of budget in Project budget and Maintenance budget boosts the accumulation of debt, Design and Architecture documentation: lack of specification/emphasis on critical architectural requirements, Reuse of Legacy / third party / open source, Parallel development, Effects Uncertainty, Non-completed Refactoring, Technology evolution, Human factor.* As found in [10], the actual number of items constantly increases over time, due to combination of several factors, including down-prioritization of ATD refactorings and the presence of unknown ATD (confirmed by our results: see *hidden ATD* as part of a *vicious circle*, III.B.2)). The implications of blending the models in this paper with the ones in [10] lead to combine the strict (linear) monotonicity of accumulating a number of ATD with items for which the interest is growing linearly or even non-linearly. The result of such combination, which is a multiplication of a linear function (the accumulation of ATD items) with a linear or potentially non-linear one (the interest accumulated for each item), leads in any case to non-linearity, and suggests that combining the accumulation of items with high interests might be catastrophic, leading to a crisis quite quickly.

## *B. Limitations*

The outcomes are the results of qualitative investigation. The results are not meant to substitute precise models derived from quantitative data, but rather to facilitate their creation. The magnitude and the proportions represented in the graphs are qualitatively formulated and may vary from context to context. We didn't aim, in this paper, at giving precise measurable results, but rather showing sociotechnical macro phenomena that represent potential threats for software development. The graphs are not supposed to be used for precise estimation as they are in this paper, but might be used to drive the collection of key data in order to build more exact models. In the field of 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 several cases to provide a more precise characterization of the phenomena. The taxonomy of ATD items might not be complete, since we focused on the most dangerous violations. Also, different contexts might show different effects for such items.

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, since we deal with real-context examples, a complex interrelation of variables that we might have not taken in consideration might have influenced the results. However, we have mitigated this problem by validating the models across 7 sites. The same holds for the third threat, *nonspuriousness*, since we don't see other alternative explanations, especially when the phenomena were repeatedly mentioned across the sites. As for external validity, although we cannot generalize, we can rely on the fact that a higher number of cases (seven) have been studied, which is higher than in many other work in literature, where usually a single or few case studies are taken in consideration. In our work, we included 7 organizations, which we might consider as a sufficient number considering the effort that we employed in gathering extensive qualitative data form each site. The last threat is the *confirmation bias*, which might happen at the validation step, when we propose the models and the respondents might have tried to fit examples because they would have liked to "believe" in the proposed model. However, many concrete examples from different cases for each phenomenon were gathered, and we always probed the statements by asking for explicit details that would confirm the legitimacy of the validation.

## *C. Related Work*

The phenomenon of ATD has only recently received the attention of the research community [3][2]. It was difficult, therefore, to find extensive previous research tackling the problem of prioritizing ATD items based on their effects in real contexts, which motivated our exploratory study. Few single case-studies were related to code-debt (e.g. [12]) or code-smell [13]. The work done by Sjøberg et al. shows that some code-smells don't create extra maintenance effort: the implications are that not all the "smells" or the architecture quality problems identified in literature have necessarily a big impact on effort: we followed such idea on an architecture level, and we empirically classified ATD creating more effort (for prioritization purposes). In the case-study conducted by Nord et al. [4], the authors studied a specific case in depth, modeling and developing a metric based on architectural rework that would help deciding between different paths leading to different outcomes. This initial attempt to create a metric made by the authors focuses on the impact of modularity in the interest to be repaid in the future. The resulting metric seems promising and, although not complete (dependencies are not "weighted") and presenting drawbacks, as the authors explain themselves, we see the opportunity for it to be further developed in order to be used for increasing the awareness of the *contagious debt* phenomenon that we have modeled here. The scope of such paper is specific for one ATD item (which falls under our *Dependency violations* class) and does not attempt to classify different ATD items based on their effects. We find an interesting point of discussion the difference between the intentional and unintentional debt, as introduced by Fowler [14], and the assumption, in more than one work (e.g. [4], [15]) that unintentional debt is linked to code debt while architectural debt is necessary strategically chosen and intentional. However, our empirically collected results and a previous study [10], in which we collected the causes for ATD items, suggest that also at the architectural level there is an accumulation of ATD that is unknown and therefore 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 [15]. However, such method only focuses on a generic quality level and its maintenance effort and not on the classification and prioritization of different ATD items, and does not take in consideration the organizational- and processes-related phenomena connected to it. Our work has been inspired by the work done by Seaman, Guo et al. [16],[8],[9]. In [16] 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 took a more holistic perspective taking in consideration a broad landscape of socio-technical phenomena as we have done in this paper, by surveying different sites. We offer the classification of dangerous ATD items and effects and we include a set of socio-technical issues that, as we have shown, might create vicious circles that go beyond the technical phenomena.

## V. Conclusions

Strategic decisions on short term and long term prioritization of Architectural Technical Debt items need to be balanced and need to rely on the awareness of the interest that needs to be paid in the future when some debt is taken. In order to estimate the interest, it's important to know the effects that such items will have in the future, especially the ones that might lead to dangerous situations. The reaching of a crisis point when the ATD is hindering the responsiveness in providing new customer value, as required in Agile, has shown to be a relevant problem that many companies struggle with. In this paper we have provided a taxonomy of the ATD items that have been found, according to a wide amount of empirical data collected in 7 sites at 5 large international software companies, the most dangerous in terms of generated effort (informing our research question RQ1). We have provided a model of the effects and the related activities (which represent the interest) that can be connected with the ATD classes in the previously mentioned taxonomy (informing RQ2). Finally, we have further analyzed the relationships in the model, discovering and modeling patterns of socio-technical events such as *contagious debt* and *vicious circles* that cause the continuous increment of interest to be paid over time, which might be linear but could potentially reach non-linearity, leading to catastrophic events such as development crisis. The goal in research and industry is to improve the practices to monitor, uncover and prioritize the dangerous ATD present in the system. Companies should focus on recognizing situations where refactorings are not completed, and in identifying key properties of ATD items such as the "contagiousness", both because dangerous for the company itself but also for avoiding pandemic spread of the ATD for the sake of entire ecosystems.

## Acknowledgment

We thank the companies that are partners of the Software Center and especially the architects group involved in the project run by the researchers authors of this article. We would also thank Michel Chaudron and Lars Pareto.

## References

[1] T. Dingsøyr, S. Nerur, V. Balijepally, and N. B. Moe, "A decade of agile methodologies: Towards explaining agile software development," *J. Syst. Softw.*, vol. 85, no. 6, pp. 1213–1221, Jun. 2012.

[2] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical Debt: From Metaphor to Theory and Practice," *IEEE Softw.*, vol. 29, no. 6, pp. 18–21, 2012.

[3] E. Tom, A. Aurum, and R. Vidgen, "An exploration of technical debt," *J. Syst. Softw.*, vol. 86, no. 6, pp. 1498–1516, Jun. 2013.

[4] R. L. Nord, I. Ozkaya, P. Kruchten, and M. Gonzalez-Rojas, "In Search of a Metric for Managing Architectural Technical Debt," in *2012 Joint Working IEEE/IFIP WICSA and ECSA*, 2012, pp. 91–100.

[5] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empir. Softw. Eng.*, vol. 14, no. 2, pp. 131–164, Dec. 2008.

[6] R. K. Yin, *Case Study Research: Design and Methods*. SAGE, 2009.

[7] A. Strauss and J. M. Corbin, *Grounded Theory in Practice*. SAGE, 1997.

[8] Y. Guo and C. Seaman, "A Portfolio Approach to Technical Debt Management," in *Proceedings of the 2Nd Workshop on Managing Technical Debt*, New York, NY, USA, 2011, pp. 31–34.

[9] N. Zazworka, R. O. Spínola, A. Vetro', F. Shull, and C. Seaman, "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, pp. 42–47.

[10] A. Martini, J. Bosch, and M. Chaudron, "Architecture Technical Debt: Understanding Causes and a Qualitative Model," in *40th Euromicro Conference on Software Engineering and Advanced Applications*, Verona, 2014, pp. 85–92.

[11] M. A. A. Mamun, C. Berger, and J. Hansson, "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.

[12] Y. Guo, C. Seaman, R. Gomes, A. Cavalcanti, G. Tonin, F. Q. Da Silva, A. L. M. Santos, and C. Siebra, "Tracking technical debt—An exploratory case study," in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, 2011, pp. 528–531.

[13] D. I. K. Sjoberg, A. Yamashita, B. C. D. Anda, A. Mockus, and T. Dyba, "Quantifying the Effect of Code Smells on Maintenance Effort," *IEEE Trans. Softw. Eng.*, vol. 39, no. 8, pp. 1144–1156, Aug. 2013.

[14] M. Fowler, "Technical Debt Quadrant," 2009. .

[15] A. Nugroho, J. Visser, and T. Kuipers, "An empirical model of technical debt and interest," in *Proceedings of the 2nd Workshop on Managing Technical Debt*, New York, NY, USA, 2011, pp. 1–8.

[16] C. Seaman, Y. Guo, N. Zazworka, F. Shull, C. Izurieta, Y. Cai, and A. Vetro, "Using technical debt data in decision making: Potential decision approaches," in *2012 Third International Workshop on Managing Technical Debt (MTD)*, 2012, pp. 45–48.