2014 40th Euromicro Conference on Software Engineering and Advanced Applications

Architecture Technical Debt: Understanding Causes and a Qualitative Model

Antonio Martini, Jan Bosch, Michel Chaudron
Computer Science and Engineering, Software Engineering
Chalmers University of Technology | Gothenburg University
Göteborg, Sweden
{antonio.martini, jan.bosch}@chalmers.se, michel.chaudron@cse.gu.se

***Abstract* — A known problem in large software companies is to balance the prioritization of short-term with long-term responsiveness. Specifically, architecture violations (Architecture Technical Debt) taken to deliver fast might hinder future feature development, which would hinder agility. We conducted a multiple-case embedded case study in 7 sites at 5 large companies in order to shed light on the current causes for the accumulation of Architectural Technical Debt that causes effort. We provide a taxonomy of the factors and their influence in the accumulation of debt, and we provide a qualitative model of how the debt is accumulated and recovered over time.**

Keywords—architectural technical debt, agile software development, development speed, software life-cycle, influencing factors, qualitative model, Grounded Theory

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 relates taking sub-optimal decisions in order to meet short-term goals to taking a financial 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). ATD is regarded as violations in the code towards an intended architecture. An example of ATD is the presence of not allowed dependencies that cause propagation of refactoring, as studied in a case study by Nord et al. [4].

ATD has been recognized as part of TD, but the specific phenomenon of accumulation of ATD and its recovery to avoid the later payment of interest (in terms of effort) due to ATD has not been tackled yet. The study of such subject would also compensate the lack, in ASD frameworks, of activities for enhancing agility in the task of developing and maintaining software architecture in large projects [5].

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

RQ1: What factors cause the accumulation of ATD?

RQ2: What are the current trends in practice in the accumulation and recovery of ATD over time?

In this paper we have employed a 1-year multiple-case embedded case-study involving 7 different sites in 5 large Scandinavian companies in order to shed light on the phenomenon of accumulation and recovery of ATD. We have analyzed the qualitative data coming from more than 30 hours of focus group interview using a combination of inductive and deductive approach proper of Grounded Theory. We have qualitatively developed and validated a taxonomy of the factors to inform RQ1 and a set of models to inform RQ2.

The main contributions of the papers are therefore:

- A taxonomy of the causes for ATD: we provide the factors for the explanation of the phenomena such as accumulation and recovery of ATD. These factors might be studied and treated separately, and offer a better understanding of the overall phenomenon.
- A qualitative model of the trends in accumulation and recovery of ATD over time. Such model:
  - Shows the strictly increasing trend of ATD accumulation and how it eventually reaches a crisis point. We connect such phenomenon to the different factors and their influence on such accumulation.
  - Helps identifying problem areas and points in time for the development of practices that would 1) avoid accumulation of ATD and/or 2) ease the recovery of ATD. Such practices would be aimed at delaying the crisis point.

II. RESEARCH DESIGN

We planned a multiple-case embedded case study involving 7 sites in 5 large software development companies. For confidentiality reasons, we will call the companies A, B, C, D and E. Companies A-D had extensive in-house embedded

978-1-4799-5795-8/14 $31.00 © 2014 IEEE
DOI 10.1109/SEAA.2014.65

software development, while company E was developing general purpose software. The choice for including company E was to compare the results with non-embedded software development. We involved 3 different units within the same company, C, and we will refer to them as C_1, C_2 and C_3. We used this approach in order to assess the variance within the same company. The companies studied were to have some years of experience of ASD. The companies chosen were situated in the same geographical area (Scandinavia), but were active on different international markets.

A. Case Description

Company A is involved in the automotive industry. Part of the development is carried out by suppliers, some by in-house teams following Scrum. The surrounding organization follows a stage-gate release model for product development. Business is driven by products for mass customization. The specific unit studied provides a software platform for different products.

Company B is a manufacturer of recording devices. Teams work in parallel in projects: some of the projects are more hardware oriented while others are related to the implementation of features developed on top of a specific Linux distribution. The software involves in house development with the integration of a substantial amount of open source components. Despite the Agile set up of the organization, the iterations are quite long compared to the other companies involved in the study.

Company C is a manufacturer of telecommunication system product lines. Their customers receive a platform and pay to unlock new features. The organization is split in different units and then in cross-functional teams, most of which with feature development roles. Most of the teams use their preferred variant of ASD (often Scrum). Features were developed on top of a reference architecture, and the main process consisted of a pre-study followed by few (ca. 3) sprint iterations. The embedded cases studied slightly differed: C3 involved globally distributed teams (Europe) while the other units (C1 and C2) teams were co-located in the same city.

Company D is a manufacturer of a product line of devices for the control of urban infrastructure. The organization is divided in teams working in parallel. The organization has also adopted principles of software product line engineering, such as the employment of a reference architecture.

Company E is a company developing software for calculating optimized solutions. The software is not deployed in embedded systems. The company has employed ASD with teams working in parallel. The product is structured in a platform entirely developed by E and a layer of customizable assets for the customers to configure. E supports also a set of APIs for allowing development on top of their software.

All the companies have adopted a component based software architecture, where some components or even entire platforms are re-used in different products. The language that is mainly used is C and C++, with some parts of the system developed in Java and Python. Company A uses a Domain Specific Language (DSL) to generate C code, while company E uses a DSL for specifying rules to be converted into libraries. The development at C3 involves extensive XML.

B. Data collection

The research design is outlined in Fig. 1.

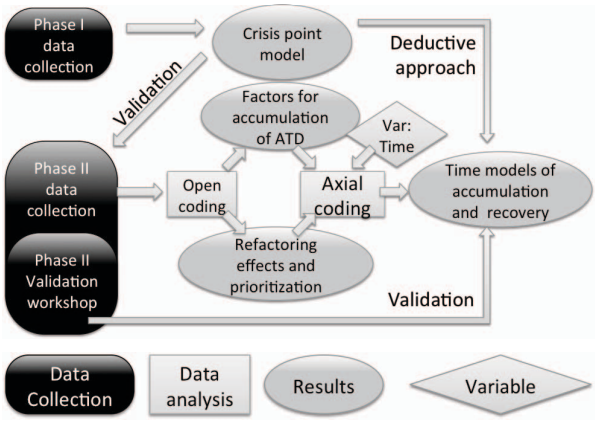

Fig. 1. Our research design

We planned a 2-phase investigation of the ATD accumulation and recovery problem. We started with a preliminary study involving 3 of the abovementioned cases, in particular A, C1, and C2, in which we explored the needs and challenges of developing and maintaining architecture in an Agile environment in the current companies. We organized three workshops at the different sites involving several roles. The workshops were to last 4 hours and were to involve developers, testers, architects responsible for different levels of architecture (from low level patterns to high level components) and product managers. In total we collected 12 hours of interactive workshop discussions involving 25 employees. The results from the first iteration were validated and discussed in a final one-day workshop involving 40 representatives from all the 7 cases, divided into two plenary discussions and a more in depth group discussion for specific topics.

The preliminary study showed a major challenge in managing Architectural Technical Debt (ATD) and its economical implications related to costs and time. In particular, we developed the crisis point model explained in III.B.1).

The second phase involved 7 workshops of 2 hours, one for each company. We involved developers, architects and in 4 cases a product owner. The total amount of data is 14 hours of interactive group interviews involving 25 employees. The formal interviews were also complemented with the study of the various architecture documentations and several hours of individual and more informal meetings with the architects responsible for the documentation.

The workshop followed a process to identify important inconsistencies that needed to be tracked because of their impact in decreasing developing time (ATD). We started with a brief introduction of what ATD is, using references from several sources included in this paper (e.g. the purpose of tracking ATD, [7]), which were also provided to the informants in advance. We took a retrospective approach, in order to identify real cases happened in the recent past rather than rely on speculations about what could happen in the future:

- we asked about major refactorings and high effort perceived during feature development or maintenance work leading to architecture inconsistencies
- we investigated the causes for the identified inconsistencies (ATD).
- we asked to explain the current process of identification of architecture inconsistency
- we asked how the ATD recovery was prioritized

The strength of this technique relies on finding the relevant inconsistencies (ATD) by starting from their causes instead of investigating a pool of all the possible inconsistencies and then selecting the relevant ones.

The second phase included a final one-day workshop were 15 representatives attended from all the various companies, providing validation of the results and follow-up discussions.

C. Data analysis

The workshops were recorded and transcribed. The analysis was done following an approach based on Grounded Theory [8] and using a tool for qualitative analysis, to keep track of the links between the codes and the quotations they were grounded to. The analysis followed the steps highlighted in Fig. 1.

Open Coding – First we analyzed the data in search for emergent concepts following open coding, which would bring novel insights on the analyzed issue. Then we applied a deductive approach using the taxonomies developed during the pre-study. The taxonomies were then updated with the newly found categories. For example, among the "causes" we added a new category called "non-completed refactoring", which was not identified in the pre-study. The results of such analysis led to the factors outlined in section III.A.

Axial Coding – The codes and categories were compared through axial coding in order to highlight connections orthogonal to the previous developed categories. Since ATD is strictly connected with time, we have used the time as axis for axial coding. This step showed the presence of patterns that were used to represent models as well as for causality and other kinds of relationships among the various inconsistencies. In our case, such analysis produced the model for accumulation and recovery of ATD explained in section III.B.

Deductive Approach – Finally, the factors and the model were deductively checked against the overall model of crisis point, developed and validated during the first phase of the research in order to understand if detailed models could fit and explain the overall one. This last analysis step showed the results in III.B.2).

D. Factors and models validation

As explained in Fig. 1, we had two step validation. The crisis point model (outlined in III.B.1) was developed during phase I. It was qualitatively validated during phase II, where *all* the informants recognized the model as representing the facts in their company. As secondary validation data, we have had several informal interviews where we have showed the model and it has always been recognized as valid.

The models for accumulation and recovery of ATD were developed in phase II and took as input the deductive model about the crisis point developed and validated previously. The new models were then validated during the second validation workshop of phase II.

III. Accumulation and Recovery of Architecture Technical Debt

We have divided the results in two parts: first we highlight the causes for ATD accumulation (factors). Then we use such factors to describe a model for accumulation and recovery of ATD over time.

A. Causes of ATD accumulation (factors)

1) Business factors

a) Uncertainty of use cases in the beginning

The previous point a) also suggests the difficulty in defining a design and architecture that has to take in consideration a lot of unknown upcoming variability. Consequently, the accumulation of inconsistencies towards a "fuzzy" desired design/architecture is more likely to take place in the beginning of the development (for example, during the first sprint).

b) Business evolution creates ATD

The amount of customizations and new features offered by the products brings new requirements to be satisfied. Whenever a decision is taken to develop a new feature or to create an offer for a new customer, instantaneously the desired architecture changes and the ATD is automatically created. This is especially happening in the current The number of configurations that the studied companies need to offer simultaneously seems to be growing steadily. If for each augmentation of the product some ATD is automatically accumulated when the decision is taken, the same trend of having more configurations over time implies that the corresponding ATD is also automatically accumulated faster.

c) Time pressure: deadlines with penalties

Constraints in the contracts with the customers such as heavy penalties for delayed deliveries make the attention to manage ATD less of a priority. The approaching of a deadline with a high penalty causes both the accumulation of inconsistencies due to shortcuts and the down-prioritization of the necessary refactoring for keeping ATD low. The urgency given by the deadline increases with its approaching, which also increases the amount of inconsistencies accumulated.

d) Priority of features over product

The prioritization that takes place before the start of the feature development tends to be mainly feature oriented. Small refactorings necessary for the feature are carried out within the feature development by the team, but long-term refactorings, which are needed to develop "architectural features" for future development, are not considered necessary for the release. Moreover, broad refactorings are not likely to be completed in the time a feature is developed (e.g. few weeks). Consequently, the part of ATD that is not directly related to the development of the feature at hand is more likely to be postponed

e) Split of budget in Project budget and Maintenance budget boosts the accumulation of debt.

According to the informants, the responsibility associated only with the project budget during the development creates a psychological effect: the teams tend to accumulate ATD and to push it to the responsible for the maintenance after release, which rely on a different budget.

2) Design and Architecture documentation: lack of specification/emphasis on critical architectural requirements

Some of the architectural requirements are not explicitly mentioned in the documentation. This causes the misinterpretation by the developers implementing code that is implicitly supposed to match such requirement. According to the informants, this is also threatening the refactoring activity and its estimation: the refactoring of a portion of code for which requirements were not written (but the code was "just working", implicitly satisfying them) might cause the lack of such requirements satisfaction.

As an example, three cases have mentioned temporal-related properties of shared resources. A concrete instance of such a problem is a database, and the design constraint of making only synchronous calls to it from different modules. If such requirement is not specified, it may happen that the developers would ignore such a constraint. In one example made by the informants, the constraint was violated in order to meet a performance requirement important for the customer. This is also connected with the previous point 1.d.

3) Reuse of Legacy / third party / open source

Software that was not included when the initial desired architecture was developed contains ATD that needs to be fixed and/or dealt with. Examples included open source systems, third party software and software previously developed and reused. In the former two cases, the inconsistencies between the in-house developed architecture and the external one(s) might pop up after the evolution of the external software.

4) Parallel development

Development teams working in parallel automatically accumulate some differences in their design and architecture. The Agile related empowerment of the teams in terms of design seems to amplify this phenomenon. An example of such phenomenon mentioned as causing efforts by the informants are the naming policy. A name policy is not always explicitly and formally expressed, which allows the teams to diverge or interpret the constraint. Another example is the presence of different patterns for the same solution, e.g. for the communication between two different components. When a team needs to work on something developed by another team, this non-uniformity causes extra time.

5) Effects Uncertainty

ATD is not necessary something limited to a well-defined area of the software. Changing part of the software in order to improve some design or architecture issues might cause ripple effects on other parts of the software depending on the changed code. Isolating ATD items to be refactored is difficult, and especially calculating all the possible effects is a challenge. Part of the problem is the lack of awareness about the dependencies that connect some ATD to other parts of the software. Consequently, there exist some ATD that remains *unknown*.

6) Non-completed Refactoring

When refactoring is decided, it's aimed at eliminating ATD. However, if the refactoring goal is not completed, this not only will leave part of the ATD, but it will actually create new ATD. The concept might be counter-intuitive, so we will explain with an example. A possible refactoring objective might be to have a new API for a component. However, what might happen is that the new API is added but the previous one cannot be removed, for example because of unforeseen backward compatibility with another version of the product. This factor is related to other two: time pressure might be the actual cause for this phenomenon, when the planned refactoring needs to be rushed due to deadlines with penalties (see 1.c) and the effects uncertainty (see 5), which causes a planned refactoring to take more time than estimated because of effects that have been overlooked when the refactoring was prioritized.

7) Technology evolution

The technology employed for the software system might become obsolete over time, both for pure software (e.g. new versions of the programming language) and for hardware that needs to be replaced together with the specific software that needs to run on it. The (re-)use of legacy components, third party software and open source systems might require the employment of a new or old technology that is not optimal with the rest of the system.

8) Human factor

Software engineering is also an individual activity and the causes for ATD accumulation can also be related to sub-optimal decision due to inexperience, careless, ignorance, error-prone situations and so on. A recurrent statement from the informant is that having documentation is not enough to avoid inconsistencies.

B. ATD accumulation and recovery models

Using the previously listed factors for ATD accumulation and the data on refactoring prioritization, we modeled the evolution of ATD over time with respect to the overall speed of adding features and to one specific release. The values in the pictures are only aimed at visualizing the trends perceived by the informants, and they don't represent any exact values. We have chosen the "function" format since it would explain the results in a more visual way.

1) Crisis-based ATD management

The current management of ATD is driven by a crisis. The informants explain that the ATD usually grows (black continuous line in the picture) until the effect makes adding new business value so slow (dashed line in the picture) that it becomes necessary to conduct a big refactoring or even rebuilding a platform from scratch. The usual approach is to wait for such an event with limited monitoring and limited

reduction ATD growth during development. In fact, the long-term improvement is considered risky invested time.

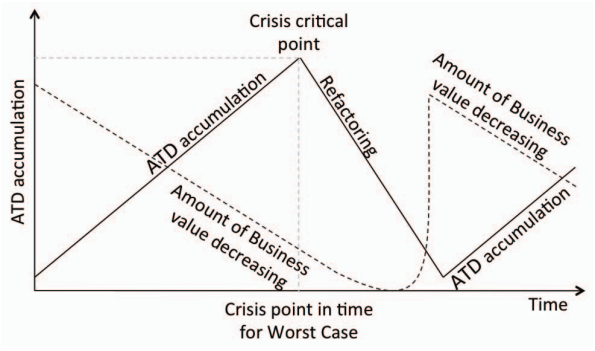

Fig. 2. ATD accumulation until the reach of a crisis point when a big refactoring is needed

2) ATD accumulation and recovery trends during feature development

In Fig. 3 we can see the various phases of ATD accumulation over time, on the left part of the graph, and the hyopthetical recovery ("complete refactory") of ATD on the right, divided by different kinds of identified ATD.

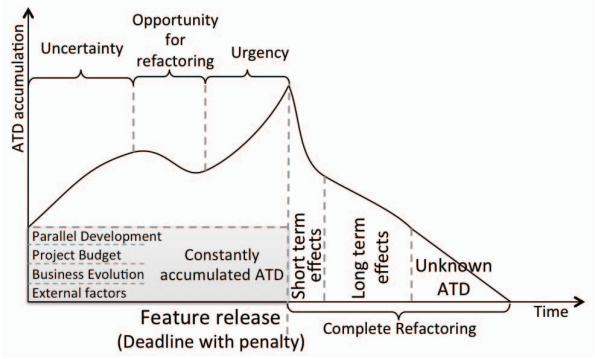

Fig. 3. Factors of constantly accumulated ATD, phases and kinds of refactorings

a) Constant ATD accumulation

From the analysis of the factors, we understood that part of the ATD is constantly accumulated over a release (grey area in Fig. 3). Such part is composed by several components described previously in section III.A: business evolution, parallel development and project budget are the one that are most connected with the companies' direct decisions, whilst other factors are external to the company (for example, the change of an Open Source module developed by third party or the tecnhology evolution). Some of them might be considered more as a "multiplier" for the other kinds of ATD, but for simplicity we treat is as a constant in the graph.

b) Phases of ATD accumulation

According to the informants, when the feature development starts there is a certain degree of uncertainty that tends to decrease over time. Since ATD is created when there is uncertainty (see section III.A.1)a) and III.A.5)), the curve on the graph representing ATD accumulation tends to raise in the beginning until the team has a more clear understanding of the requirements, desired design and desired architecture altogether. At this point, the hypothesis is that ATD accumulation slows down. The ATD starts being accumulated abundantly when the urgency for meeting the deadline shows up in the team. Urgency seems to grow constantly with the deadline approaching, causing the level of ATD to grow accordingly. We don't know exactly from the data when uncertainty stops and urgency starts and if the two phases overlap. Some informants mention a time window when the team recovers part of the ATD needed to deliver the feature, but it's unlikely that all the accumulated ATD is recovered during this phase (especially the constant one). However, this seems to be a good opportunity in the process when the team might decide to take care of the ATD before the release. The Project Budget factor might have a negative impact on such practice though.

c) Refactoring and its prioritization

Once the feature is released, there is ATD left in the system. The ideal case is that the ATD is completely removed by the system. However, this is not done or even possible according to our data, for two main reasons: part of the ATD is currently not known (see "effects uncertainty" in section III.A.5)) and the refactoring is usually only partially prioritized.

Prioritization of the refactoring depends usually on the kind of refactoring: the refactoring needed for easing (or especially allowing) the short-term release of features is usually prioritized and performed by the team. This is possible both because of the immediate need of it and because such ATD is possible to be recovered in quick time and therefore can just be included in the successive feature development. These characteristics are represented in the graph by the steep slope in the curve in correspondence to "short term effects". As for the "long term effects" refactoring, usually it's represented by some extensibility or maintainability mechanism at a more higher level of abstraction that has not been implemented during the development of the feature. To introduce such mechanism the needed time is usually substantial compared to the feature development time: for example, if the refactoring is estimated to be 2 months and a new feature is supposed to take the same amount of time to be developed, it doesn't make sense to include it into the feature as a story. Also, such task would probably influence other parts of the software, which might cause interruptions on other teams' work. For such reasons, such ATD is usually down-prioritized in favor to feature release (as explained in section III.A.1)d)).

In Fig. 4 is represented the case in which short term and long term ATD is recovered. We can see how the constant ATD and the delay effect from the unknown ATD are continuously accumulated even in the case all the possible ATD is recovered.

In Fig. 5 we show the case in which there is no refactoring performed before the feature is started. Even though the feature might be released earlier with respect to the previous case, when all possible refactoring was performed, the delay caused

by the ATD grows because of the addition of the unknown and the short-term ATD.

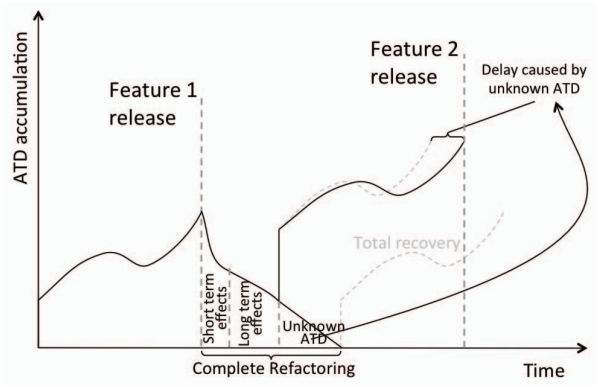

Fig. 4. Some unknown ATD is always accumulated and impacts next development: total recovery is ideal, the best option is partial recovery.

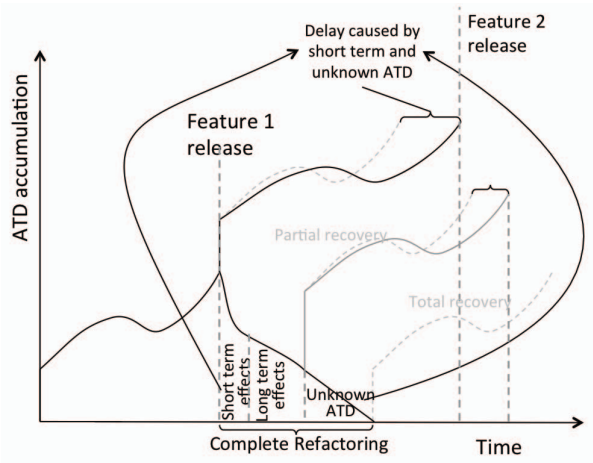

Fig. 5. No recovery is affected by short term and unknown ATD, even if the feature might be released earlier than with partial recovery.

In Fig. 6 we have represented the increasing in the delay caused by the ATD with "long term effect" with respect to the time for the partial recovery function.

We have projected the same trend for three features and the result is shown in Fig. 7. At a certain point in time, the number of released features for the two functions "No recovery" and "Partial recovery" will be the same but in case of "No recovery" there is more ATD accumulated. This means that such ATD either needs to be partially eliminated (which requires time) in order to reach the same status of "Partial recovery", or the delays will continue to grow. In the latter case "Partial Recovery" would become constantly more convenient. Notice that the number of features released before the crisis point might vary according to the magnitude of the variables involved. However, the trend would remain the same.

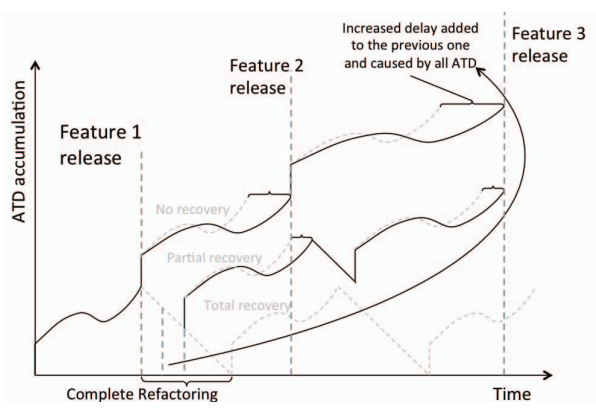

Fig. 6. After a number of features released the long term ATD starts to have an impact.

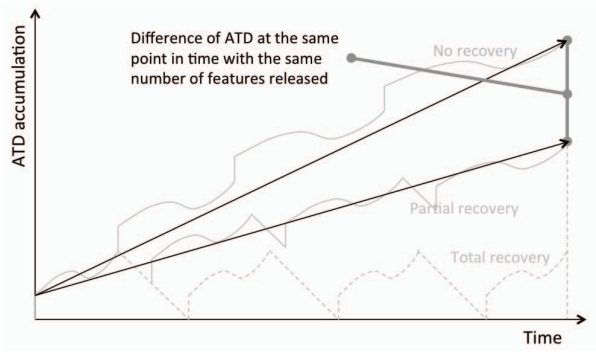

Fig. 7. Given the trend of ATD accumulation, after some feature released, the time elapsed will be the same but the ATD present in the system will be more.

IV. Discussion

The results are meant to show the current factors and trends shaping the phenomena of ATD accumulation and recovery. The results are not meant to substitute precise models derived by quantitative data, but rather to facilitate their creation. For example, the magnitude and the proportions represented in the graphs are qualitatively formulated and may vary from context to context. They 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 a more exact model. 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.

The provided qualitative representation shows factors and trends that reveal, in our opinion, important implications in the light of the recently emerging practice of ATD management. One very important variable to be taken into consideration is time, and we have done a first step in order to explain the relationship between such impacting variable and the phenomena of ATD accumulation and recovery. In the followings we therefore discuss a number of implications (expressed in the form of propositions) that can be inferred by our results. They represent hypotheses qualitatively tested through a substantial number of experts from similar domains,

but that need to be quantitative complemented and assessed in the future.

a) Implications (propositions) for research and practice

It's not realistic to think that the accumulation of ATD might be completely recovered. The best option is to reduce it.

As we can see from Fig. 3 there is a constant accumulation of ATD for several reasons, some of which are also external to the company. In conjunction with this, part of the ATD remains unknown. These two factors together lead to the consequence that each iteration brings a quantity of ATD in the system, and that part of it will remain. Even if the magnitude of such accumulation is not yet clear, the function over time is obviously monotone. This fact is also confirmed by the crisis model (Fig. 2): a point of non-sustainable development, where the ATD accumulated is high, is constantly reached by the companies, which confirms the monotonicity of the trend.

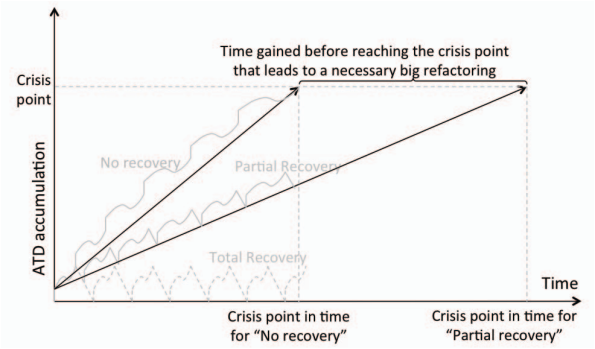

Fig. 8. Given the trend of inevitable accumulation of ATD, a crisis point will eventually be reached. The gain for the company is to reduce the number of crisis and therefore the number of costly refactorings.

The goal of partially recovering ATD is that the company can gain time before the crisis point.

Fig. 8 shows how the goal of managing ATD is not to avoid crisis but rather to gain time. The objective is still relevant, since platform creation and/or huge architectural refactorings are long and costly activities. However, the main goal for a software company cannot be to avoid such activity completely and aiming at having an "eternal" system, but rather to reduce the number of times such activity is repeated before the retirement of the product. This means that the company needs to proactively plan the activity as part of the product development.

Decreasing the amount of unknown ATD is necessary to make effective the best option in terms of ATD reduction (what we have called "Partial recovery")

As we can see in Fig. 4 the quantity of unknown ATD influences the ability to recover and therefore to keep the ATD as low as possible. This would delay the eventual crisis point. It's therefore of utmost importance for the company to establish a practice for iterative ATD discovery and to maintain ATD as a visible quality through some kind of artifact (for example (for example, as explained by Guo and Seaman, [9]).

By reducing (creating) disincentives (incentives) we can reduce (increase) the constant ATD accumulation.

Some of the factors have a disincentive (or incentive) effect on ATD accumulation, which we interpret as a tendency or probability to lead to have more ATD. In the view of ATD accumulation as a function, such tendency would represent a constant that increases the steepness of the function. Such steepness would make the approach of the crisis point faster (or slower). The known disincentives are mainly the ones described in section III.A.1) and related to business factors, such as having too much focus on features with respect to the product, having a split budget for project and maintenance and having high penalties on deadlines. To our current knowledge, such disincentives don't influence some specific ATD issues. However, future studies could increase the understanding on such matter. Incentives for avoiding accumulation are for example having described temporal properties in the documentation. Incentives to recover ATD are the prioritization of refactorings (especially missing currently are the long-term ones) and the employment of a window for creating opportunities for refactoring approximately in the middle of the feature development Fig. 3, when the refactoring seems to have more sense, and after release, as recommended also by Agile practices.

Agile software development brings advantages and disadvantages to ATD accumulation

The employment of Agile Software Development seems to bring both advantages and disadvantages to the phenomenon of ATD accumulation. Such influences are related to the incentives and disincentives previously mentioned. For example, the Agile process and principles favor the focus on the features over the product and boosts the accumulation of ATD, relying on a mandatory subsequent refactoring that is not always recognized from the management point of view. On the other hand, Agile provides an iterative process for gradually taking care of uncertainty and would allow to iteratively keep track of the ATD. Therefore, a set of lightweight practices for ATD management is needed and would benefit from the Agile iterative process if well embedded. We are currently studying such practice development by employing action research at the companies involved in the results.

b) Future work

Some open issues that require further investigation are:

- It remains unclear when is best to conduct a refactoring. The unknown effects of combining the refactoring with uncertainty and urgency suggest further investigation.
- We have studied principally the delays that the ATD have on long term development. What needs to be more investigated are the short-term benefits that are gained by the accumulated ATD.

c) Limitations

The graphs in this paper are not meant to represent precise data coming from a measurement system. Therefore the steepness of the curves and the projections might vary in real context. The magnitude for the contribution of each factor is also to be further assessed. However, we offer the recognition

of factors that are not necessarily possible to be measured and therefore discovered by quantitative analysis, such as urgency and uncertainty. The results are qualitatively developed through a thorough research process and using a wide amount of qualitative data coming from more than 30 informants from 7 sites and with different roles, which allowed as to compare and test statements among themselves. Furthermore, architecture documentation has been evaluated as well as secondary data. This has allowed us to apply source triangulation [10].

d) Related Work

Lehman et al. [11] propose a formal approach for process modeling. The paper emphasizes the usefulness of formal models (e.g. functions) for effort prediction. We have developed the crisis model (Fig. 2), which can be considered the abstract model, and we have done a first calibration by finding the factors that are needed to describe the formal function (as parameters of the function). Our approach can be considered as a first necessary step towards the formalization and the precise prediction of the process, which needs more quantitative data. An empirical model of debt and interest is described by Nugroho et al. [12]. However, such method only focuses on the interest paid during maintenance and it's not focused on finding the causes for the accumulation of debt. The business factors that we have found are missing in the study of ATD as also reported from a single case study [13]. Sindhgatta et al. [14] have studied the software evolution in an Agile project, where some of the Lehman laws were tested through project sprints. Some of the results suggest a confirmation of the trends that we have identified. For example, the laws of (continuous) change and growth show the monotonicity of system growth and the necessity for the system to adapt to the business environment, which are recognized also in our factors. However, the results are not directly connected with ATD.

In summary, the only aggregation of information provided by different sources is the paper by Tom et al.[3], where ATD was just discovered as part of the TD while the other studies are related to single case studies. Most of the studies don't either consider generic TD and not ATD or don't focus on finding the causes for ATD accumulation.

V. CONCLUSIONS

Decisions on short term and long term prioritization of architecture improvements need to be balanced and need to rely on the knowledge of the underlying phenomenon of ATD. The reaching of a crisis point when the ATD is hindering the responsiveness in providing new customer value, as required in ASD, has shown to be a relevant problem that many companies struggle with. Such crisis point seems to be inevitable given the continuous accumulation of ATD and the impossibility to recover all of it. However, the act of slowing down the ATD accumulation would reduce, in the long term, the number of crisis points when a huge costly refactoring or the replacement of the whole system need to be conducted. We have shown such accumulation and recovery trends over time in order to inform RQ1. We have shown what are the causes of the accumulation of ATD, and we outline, through the recognition of different influencing factors, clear objectives that can be treated or further studied in order to avoid or mitigate the accumulation of ATD, which informs RQ2. An important goal in research and industry is to improve the practices to uncover ATD present in the system. It's also important to identify the best points in time for performing refactoring and therefore repaying the debt that is going to generate more interest effort later on. Such practices need to complement the current Agile process in place, in order to keep responsiveness stable through the whole software development process.

ACKNOWLEDGMENT

We thank the companies that are partners of the Software Center and participating in the project run by the research team composed by the authors of this article. We are also extremely grateful for the work done by Lars Pareto, member of the team and participating in the project until 2013.

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] M. Daneva, E. van der Veen, C. Amrit, S. Ghaisas, K. Sikkel, R. Kumar, N. Ajmeri, U. Ramteerthkar, and R. Wieringa, "Agile requirements prioritization in large-scale outsourced system projects: An empirical study," *J. Syst. Softw.*, vol. 86, no. 5, pp. 1333–1353, May 2013.

[6] U. Eklund and J. Bosch, "Applying Agile Development in Mass-Produced Embedded Systems," in *Agile Processes in Software Engineering and Extreme Programming*, Springer, 2012, pp. 31–46.

[7] 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.

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

[9] 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.

[10] 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.

[11] M. M. Lehman, G. Kahen, and J. F. Ramil, "Behavioural Modelling of Long-lived Evolution Processes: Some Issues and an Example," *J. Softw. Maint.*, vol. 14, no. 5, pp. 335–351, Sep. 2002.

[12] 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.

[13] 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.

[14] R. Sindhgatta, N. C. Narendra, and B. Sengupta, "Software Evolution in Agile Development: A Case Study," in *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, New York, NY, USA, 2010, pp. 105–114.