THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

AMBIDEXTERITY IN LARGE-SCALE SOFTWARE ENGINEERING

ANTONIO MARTINI

Department of Computer Science and Engineering
Division of Software Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY

Göteborg, Sweden 2015

Ambidexterity in large-scale software engineering

ANTONIO MARTINI

ISBN 978-91-7597-285-5

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie Nr 3966
ISSN 0346-718X

Technical Report 121D
Department of Computer Science and Engineering
Division of Software Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Sweden
Telephone + 46 (0)31-772 1000

Printed at Chalmers Reproservice
Göteborg, Sweden 2015

To Ildikó and my family

ABSTRACT

Software is pervading our environment with products that become smarter and smarter every day. In order to follow this trend, software companies deliver continuously new features, in order to anticipate their competitors and to gain market share. For this reason, they need to adopt processes and organization solutions that allow them to deliver continuously.

A key challenge for software organizations is to balance the resources in order to deliver enough new features in the short-term but also to support the delivery of new features in the long-term. In one word, companies need to be ambidextrous. In this thesis we investigate what ambidexterity is, what are the factors that hinder large software companies to be ambidextrous, and we provide initial solutions for the mitigation of such challenges.

The research process consists of an empirical investigation based on the Grounded Theory approach, in which we conducted several case studies based on continuous interaction with 7 large software organizations developing embedded software. The results in this thesis are grounded in a large number of data collected, and corroborated by a combination of exploratory and confirmatory, as well as qualitative and quantitative data collection.

The contributions of this thesis include a comprehensive understanding of the factors influencing ambidexterity, the current challenges and a proposed solution, CAFFEA. In particular, we found that three main challenges where hampering the achievement of ambidexterity for large software companies. The first one is the conflict between Agile Software Development and software reuse. The second one is the complexity of balancing short-term and long-term goals among a large number of stakeholders with different views and expertize. The third challenge is the risky tendency, in practice, of developing systems that does not sustain long-term delivery of new features: this is caused by the unbalanced focus on short-term deliveries rather than on the system architecture quality. This phenomenon is referred to as Architectural Technical Debt, which is a financial theoretical framework that relates the implementation of sub-optimal architectural solutions to taking a debt. Even though such sub-optimal solutions might bring benefits in the short-term, a debt might have an interest associated with it, which consists of a negative impact on the ability of the software company to deliver new features in the long-term. If the interest becomes too costly, then the software company suffers delays and development crises. It is therefore important to avoid accumulation, in the system, of Architectural Technical Debt with a high interest associated with it.

The solution proposed in this thesis is a comprehensive framework, CAFFEA, which includes the management of Architectural Technical Debt as a spanning activity (i.e., a practice shared by stakeholders belonging to different groups inside the organization). We have recognized and evaluated the strategic information required to manage Architectural Technical Debt. Then, we have developed an organizational framework, including roles, teams and practices, which are needed by the involved stakeholders. This solutions have been empirically developed and evaluated, and companies report initial benefits of applying the results in practice.

ACKNOWLEDGMENTS

I would like to thank all the people that have made this work possible and my journey a great experience. They are too many to fit here, but I thank them all and their involvement will always be engraved in this thesis.

First of all, I would like to thank my supervisor, Professor Jan Bosch, for giving me the opportunity to conduct this research. Jan has given me trust and freedom from the beginning, and he has always supported me and believed in my work. I have learnt a lot from his experience and his determined way of seeking novel and ground-breaking contributions to the field of Software Engineering. Jan, as the director of the Software Center, has provided me with a great environment for conducting high quality empirical research, and has shown me how to make possible the successful interaction between academia and industry. I also thank Jan for always giving his contribution to the publications, despite, sometimes, my last-minute submissions.

I want to thank also Lars Pareto, my second supervisor for the first 2 years of PhD. Before Lars unexpectedly and tragically passed away, he coached and supported me with true dedication. Lars contributed to my growth with his experience in applying a thorough methodology and his encouragement to problematize knowledge rather than accepting common sense for truth: both have deeply influenced my approach to the research reported in this thesis. Lars was also very important for my successful interaction with the industrial partners of the Software Center, sharing his experience, his contacts and his previous work from which I have taken inspiration.

I would also like to thank my second supervisor, Associate Professor Helena Holmström Olsson, for her kind and patient review of the many pages included in this thesis. Her point of view helped me looking at my whole work with a new and holistic perspective. I am also grateful to Professor Michel Chaudron, who stepped in supervising me when Lars Pareto passed away, and supported me with his knowledge and new ideas for my research. Also a special thank to my examiner, Magnus Bergquist, who has always provided me with very a responsive and extensive feedback.

A special thank to the Software Center for funding this research project, and to all the companies that have been involved: the main contacts, who made this research possible, the many development teams involved and the architects, who especially contributed with their experience, knowledge and active work to this thesis. Just to mention a few of them, I will mention: Jonas Holmgren and Fredrik Hugosson at Axis, Jørgen Karkov, Eva Nielsen and Niels Jørgen Ström at Grundfos; Jesper Derehag, Anders Dyvermark, Staffan Enhebom, Peter Eriksson, Henrik Harmsen, Peter Kanderholm, Anders Kvist, Mats Lindén, Richard Lundberg, Pat Mulchrone, Anna Sandberg and Björn Östlund at Ericsson; Peter Sutton at Jeppesen; Jesper Lindell, Sven Nilsson, Christoffer Höglund and Vilhelm Bergman at Saab; Jens Svensson and Andreas Henning at Volvo AB; Kent Niesel, John Lantz and Ilker Dogan at Volvo Cars.

Many thanks to all my colleagues at the Software Engineering Division, where I have found both friendship and great discussions. A special thank to Dr. Ali Shahrokni, whose friendship has extended after his PhD; to one of the best office-mates that I've ever had, Hiva Alahyari and to Dr. Emil Alegroth for the great discussions "on the road", Ulf Eliasson, Vard Antinyan and Abdullah Al Mamun for their collaboration on the Technical Debt topic and to all the other PhD candidates at the division and the departments of CSE, current and past, with whom I have shared tons of interesting discussions, laughs and fun activities. Many thanks also to all the seniors of the division, especially to Associate Professor Patrizio Pelliccione, for the great

discussions on research, teaching and many other topics, Professor Robert Feldt for the always interesting input on research quality, to Associate Professor Eric Knauss for discussing ideas and reviewing papers even very late at night, and to Associate Professor Christian Berger for his constant interest in my research and for sharing his knowledge on research opportunities. Thanks also to all the fellow researchers outside this university: especially Senior Lecturer Romina Spalazzese, Senior Lecturer Ulrik Eklund, and special thanks to Senior Lecturer Lars Bendix, who has been my Master Thesis supervisor and who supported me in the choice of pursuing this PhD.

The most important thank is perhaps for my amazing and supportive life-companion, Ildikó: nothing would have been possible without her, who was there in the best and the worst moments of this journey. Since she is also involved in a very similar academic experience, I hope that I will be as important for her success as she has been for mine.

Another special thank to my family: my father Giuseppe, my mother Maria Luisa and my sister, Francesca, who have always been supporting me in any possible situation. A big thank also to all the rest of the family, who has always been there for me whenever the work allowed me to visit my beloved Italy. I cannot thank all my friends enough for always making my life interesting also outside work. A final thank to my pet fish, who have made me company while writing this thesis, and to their 14 new-born, who symbolize how this is not the end of a journey, but perhaps just its beginning.

LIST OF PUBLICATIONS

INCLUDED PAPERS

1. Martini, A., Pareto, L., Bosch, J. *"Enablers and Inhibitors for Speed with Reuse"*, published in proceedings of Software Product Lines Conference, SPLC 2012 [71].
2. Martini, A., Pareto, L. & Bosch, J., 2013 *"Improving Businesses Success by Managing Interactions among Agile Teams in Large Organizations"*, published in the proceeding for 4th international conference in software business (ICSOB 2013) [91]
3. Martini, A., Pareto, L. & Bosch, J., 2014. *"A multiple case study on the inter-group interaction speed in large, embedded software companies employing Agile"* accepted in Journal of Software: Evolution and Process (in press)
4. Martini A., Bosch J., and Chaudron M. *"Investigating Architectural Technical Debt Accumulation and Refactoring over Time: a Multiple-Case Study"* Information and Software Technology, in press [121].
5. Martini A., Bosch J.: *"Contagious Technical Debt and Vicious Circles: a Multiple Case-Study to Understand and Manage Increasing Interest"* submitted to
6. Martini A., Bosch J.: *"Towards Prioritizing Architecture Technical Debt: Information Needs of Architects and Product Owners"* Accepted for publication in proceeding of Euromicro SEAA 2015 [130]
7. Martini A., Pareto L., and Bosch J., *"Towards Introducing Agile Architecting in Large Companies: The CAFFEA Framework,"* in Agile Processes, in Software Engineering, and Extreme Programming, 2015. [135]

OTHER PAPERS

1. Martini, "Factors influencing reuse and speed in three organizations," Technical Report GUPEA, 2012.
2. A. Martini, "Managing Speed in Companies Developing Large-Scale Embedded Systems.," in *Proceedings of ICSOB*, Potsdam, Germany 2013
3. A. Martini, L. Pareto, and J. Bosch, "Communication factors for speed and reuse in large-scale agile software development," in *Proceedings of the 17th International Software Product Line Conference*, Tokyo, Japan, 2013
4. A. Martini, J. Bosch, and M. Chaudron, "Architecture Technical Debt: Understanding Causes and a Qualitative Model," in *Proceedings of Euromicro SEAA*, Verona, Italy, 2014
5. A. Martini, L. Pareto, E. Knauss, and J. Bosch, "Boundary-spanning activities in large embedded software companies employing Agile Software Development." *In Proceedings of IRIS38,* 2014
6. A. Martini, L. Pareto, and J. Bosch, "Role of Architects in Agile Organizations," in *Continuous Software Engineering*, Ed. Springer International Publishing, 2014
7. A. Martini, L. Pareto, and J. Bosch, "Teams interactions hindering short-term and long-term business goals," in *Continuous Software Engineering*, Springer International Publishing, 2014

8. A. Martini, L. Pareto, and J. Bosch, “A Framework for Speeding Up Interactions Between Agile Teams and Other Parts of the Organization,” in *Continuous Software Engineering*, Springer International Publishing, 2014
9. A. Martini and J. Bosch, “The Danger of Architectural Technical Debt: Contagious Debt and Vicious Circles,” in *2015 12th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2015
10. U. Eliasson, A. Martini, R. Kaufmann, and S. Odeh, “Identifying and Visualizing Architectural Debt and Its Efficiency Interest in the Automotive Domain: A Case Study.” In *Proceedings of International Conference on Software Maintenance and Evolution,* Bremen, Germany, 2015.
11. B. Vogel-Heuser, S. Rösch, A. Martini, and M. Tichy, “Technical Debt in Automated Production Systems,” *Proceedings of International Conference on Software Maintenance and Evolution,* Bremen, Germany, 2015.

PERSONAL CONTRIBUTION

For all publications above, the first author is the main contributor. In all publications appended in this thesis, I was the main contributor with regard to inception, planning and execution of the research, and writing. The same applies for the additional publications in which I am listed as first author.

For the two publications in which I am listed as co-author, the following contributions were made by me:

- Eliasson et al.: the study was conducted by two Master Students at Volvo Cars. Ulf Eliasson was the main coordinator for the conduction of the study inside the company. I initiated the study, proposed the design and supervised the study from a Technical Debt perspective, driving the research choices and I took part in the writing.
- Vogel-Heuser et al.: the study is a vision paper about introducing the term Technical Debt into APS system, including not only software development, but also mechanical and electrical engineering. I contributed with the Technical Debt perspective on the reported cases and with my knowledge of the existing related work on the subject.

TABLE OF CONTENT

1 INTRODUCTION

1.1 PREFACE

Software is becoming omnipresent in our environment, making everything "smarter" and "smarter" every day. The advances in many scientific fields, including software engineering, are contributing to create new features that are enriching our everyday life. For example, I remember when a few years ago I was driving on the highway and I received an important call to pick up a package that I was expecting. I stopped the car, but I answered the call too late. In that moment, I wished that I could have answered the call with just my voice. As if someone had heard me, cars have recently introduced voice control to interact to the dashboard, making easier for the driver to activate functions and to monitor parameters without using hands ("handsfree" features).

Many of this kind of features require software companies to develop and deploy software on an existing physical product. The more and interesting features a company is able to provide, the more attractive the product becomes for the customers. Consequently, companies focus on delivering features as quick as possible in order to release their products before the competitors, in order to win a market share (i.e. convince a large portion of customers to buy their products). This competition is won by the company that is able to understand the customers' needs and is therefore able to deliver new features fast. Such ability is called responsiveness.

However, sometimes software companies make short-term choices, in order to quickly develop a feature, that might hinder how a company will be able to ship another feature in the future. There are three alternative scenarios that might happen:

- A company is only focused into delivering new features as fast as possible, without considering long-term responsiveness: the company might have a good business in the short term, but in the future they might not be able to deliver to the customers what they want because they focused only on short-term needs, and therefore they might lose market share in the long run.
- A company is only focused in delivering features to the customers in the long term: the company invests only in the future, without selling new products in the short term. However, as a result, the company would lose market share and it might not be able to anticipate what the customers *will* want in the long term.
- A company is focused in delivering features to the customers in the short term, but it also invests *some* resources in order to be able to deliver features in the long term. This way, the company has good chances of winning the market share in the short term, but it is prepared for the same long-term goal.

In this thesis we try to understand how large software companies can achieve the third scenario. In such case, we say that the software company is *ambidextrous*. Such term means that the company is able to manage two different and conflicting tasks: being responsiveness in the short term but investing enough resources in order to be able to be responsive in the long term.

1.2 OVERALL FRAMEWORK

In this section we present the overall outline of this thesis. First we describe the research problem and how we have defined it in the first part of the thesis, while we have investigated a possible solution in the second part, as highlighted respectively in Figure 1 and Figure 2. The complete theoretical and empirical backgrounds behind the research questions and the contributions will be elaborated in detail in the next Chapter.

1.2.1 Research Problem

Software companies develop a product (for example, a car and its software) or a service (for example, a website), which is then used by one or more customers according to their needs. Companies need to be *competitive*, or else they need to have enough customers with respect to their competitors (i.e. have a good enough *market share*) in order to have enough revenues to cover the development costs and also to obtain a gain from the initial development investment (this measure is also called *Return on Investments*, RoI).

One way to assure a good market share and therefore a good RoI is for software companies to make available a product or service to the customers before the competitors. Thus, one of the main business goals for software companies is to develop a product or a service with new and attractive features and deliver it quickly: in this case we say that companies achieve a short time-to-market. This is a *short-term business goal*, since the companies want to gain competitiveness in the short-term.

After a product or a service is released and a market share is gained, the software company needs to keep being competitive. This can be done by releasing additional products or services (for example, products customized for a specific customer). Also, new attractive features requested by the customers or the market can be added to existing products or services. However, also in this phase it is important for the software company to keep a short time-to-market, which would assure a good enough market share by anticipating the competitors. The means for achieving such goal is to reuse existing solutions rather than developing new ones. But in order to do this, companies need to develop software systems that have certain qualities (for example, they are flexible enough to satisfy future use cases) in order to allow them to be reused or evolved. This is a *long-term business goal*, since the benefits of achieving qualities come in handy in the long-term, after the system has already been delivered to the customers for the first time. However, such qualities of a system are usually achieved during its development, and they are costly to change once the system is in place.

In summary, software companies need to fulfill both short- and long-term business goals to *become* and *remain* competitive. The current main challenge for software companies is to balance, during the development of the system, the focus between one kind of business goals and the other. In practice, the organizations need to deliver a system fast but with enough quality to support the same ability of being fast in the future. As a consequence of the lack of balance, there are situations in which the focus is put *either* on short-term *or* long-term business goals. However, focusing too much on one kind of goals only and neglecting the other one is risky, since such choice leads the software company to not being competitive in the short- or long-term. In other words, the *unbalance* between short-term and long-term business goals leads to lack of competitiveness (bottom part of Figure 1). Therefore, software companies need to *balance* short-term and long-term goals in order to remain continuously competitive (such ability is also regarded as *ambidexterity* better explained in section 2.1).

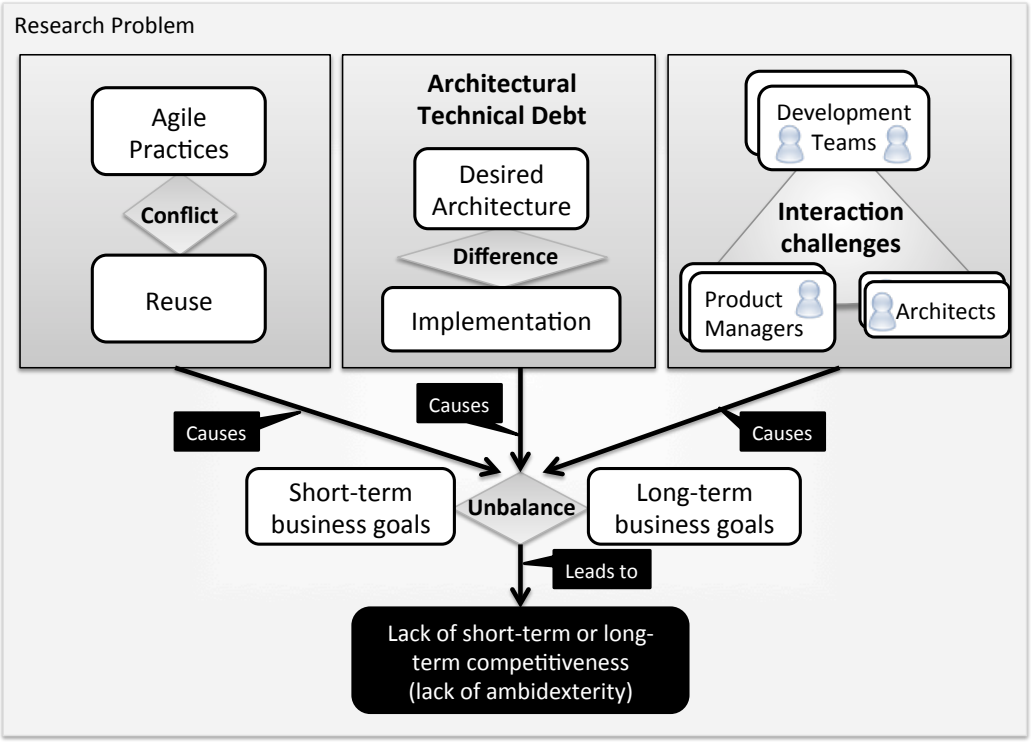


Figure 1 Research problem: three different factors cause the unbalance of short-term and long-term business goals in a software company, which in turn forces the company to be focused on either short-term or long-term competitiveness. Such factors are the conflict between Agile Practices and Reuse, the presence of Architectural Technical Debt (which is the gap between a desired architecture and the actual implementation) and the presence of interaction challenges among three key actors interacting in large-scale software development: development teams, product managers and architects.

However, how to manage this balance is not well understood, scientific literature lacks a solid theory and large software companies struggle in practice. Only recently the concept of ambidexterity, the ability of combining conflicting business goals, has been explored in the research fields of management and information systems, but not in software engineering. This gap motivated the research conducted in this thesis.

In the first part of this thesis we focused on better understanding this research problem. In fact, there might be many factors that influence the balance of short-term and long-term business goals inside a large software company. Among many factors (investigated in Chapter 4), three main factors were causing unbalance between short-term and long-term business goals. They are visible in Figure 1, and they are briefly explained below; they will be explored more in depth (since each of them include many sub-factors) in the following of this thesis.

The first factor is the conflictual relationship between Agile software development and the need for software reuse. The former one is a process recently introduced in industry that aims, among other things, at shortening the delivery of features in the short-term and focuses the development effort on quickly delivering value to the customer. The latter is the need of reusing existing software in order to lower the development and maintenance cost and decreasing the time-to-market of a new or evolved product (a long-term business goal). Chapter 4 investigates this conflict in a holistic and exploratory way, while Chapters 5 and 6 focus on a specific underlying aspect of such unbalancing conflict, namely the interactions among different parts of the organization that lead to the unbalance (as explained below as the second factor).

The second factor is related to the difficulty of balancing short- and long-term business goals in large software companies where several employees have different views and kinds of expertize. These roles are interconnected and the outcome of their

interaction becomes quite complex. Besides, not all the stakeholders (consciously or not) might act to reach the same business goal, or the combined outcome of several stakeholders' goals result as unbalanced. This is caused by several *interaction challenges* present among these groups of employees. The complexity of such environment and therefore the ability to control the outcome in order to reach balanced business goals, further increases when software development is combined with the development of a physical product, such as cars, telecommunication systems, phones, etc. In Chapter 6 we show how the multitude of interactions between a software development team and the rest of the organization influence the balance of short-term and long-term business goals. The study shows how especially critical are the interactions between the development team, the product managers and the system and software architects for balancing short-term and long-term business goals.

The third factor is related to the relationship between the short-term delivery of a system and its long-term qualities: the system needs to be delivered to the customer fast, but it also needs to support the organization to deliver new solutions in the long-term (features or products). Thus, although it is important to maintain competitiveness with short-term deliveries, it is also important that the system would satisfy long-term qualities: the means for technically achieving this is to have a desired (target) system and software architecture that supports both short-term delivery and long-term qualities. However, there is a common phenomenon that causes the business goals to be unbalanced: in some cases, the system might be developed to achieve fast delivery only, neglecting qualities related to the desired architecture, which in the long-term might cause a delayed or prevented delivery. Such phenomenon is called *Architecture Technical Debt* (ATD, explained more in details in section 2.3): this concept is based on a financial metaphor, which relates developing a sub-optimal architectural implementation (a system that does not satisfies certain desired qualities) to taking a "debt" for achieving short-term goals; however, such debt needs to be repaid with extra costs later on, which represents, in the metaphor, the interest of the debt. The consequence of accumulating too much ATD in order to achieve short-term goals might hinder the long-term ones (causing the unbalance between such business goals). This phenomenon is studied in the second part of the thesis, especially in Chapters 7 and 8. This third factor is also related to the previous one: in fact, one of the main interaction challenges among key stakeholders inside a large software company (product managers, system and software architects, developers) is related to the prioritization of short-term feature delivery against the allocation of development resources to avoid (or refactor) ATD, and therefore maintain a desired architecture to support long-term deliveries.

Figure 1 shows how the previous three factors are related to the unbalance between short-term and long-term business goal. Such unbalance creates a lack of competitiveness either in the short-term or in the long-term. This represents the research problem that we have investigated in this thesis. Thus, the related high level research question that we want to address is:

RQ_A: How can short-term and long-term business goals be balanced?

In relation to the concept of ambidexterity, this RQ can also be expressed as:

RQ_B: How can software companies be more ambidextrous?

In the next section we explain what solution was investigated in the second part of the thesis to mitigate the research problem. Also, in Chapter 2 we will explain the background in depth with more refined research questions and the link between them and the rest of the thesis (see Table 1).

1.2.2 *Solutions and evaluation*

In order to investigate a *solution* to the research problem, it was important to understand how to treat the three factors explained above (Figure 1): mitigating the conflict between Agile and reuse, mitigating the interaction challenges and managing the accumulation of sub-optimal architectural solution (Architecture Technical Debt, ATD). As explained earlier, these three factors are related: avoiding architectural technical debt, meaning avoiding sub-optimal architectural solution that have a costly negative impact in the long-term would allow software to be reused or evolved. However, the accumulation of ATD, and therefore the unbalance towards short-term goals, is due to the lack of a suitable practice that would allow the developers, architects and product managers to effectively interact in order to balance the prioritization of short-term deliveries with the ATD avoidance. We therefore investigated a practice that would improve their interactions by managing the prioritization of ATD.

In the current scientific literature, a generic solution for mitigating generic interaction challenges is called *spanning activity* (more details in section 2.4). This is a practice carried out by the groups involved in the interaction challenge. For our specific research problem, the second part of the thesis is dedicated to the development and evaluation of a spanning activity for the management of ATD. The aim of this practice is to better balance short-term and long-term business goals among the key stakeholders (product managers, architects and developers), in order for the software company to be competitive both in the short- *and* long-term, which makes the company ambidextrous, as explained in Figure 2.

The spanning activity of ATD management includes the management of several sub-activities: collecting information about ATD (which usually involves architects and developers, respectively the experts in the architecture and in the implementation), understanding the potential risk to pay a high interest on a debt and the prioritization of allocating resources to repay such debt (which includes the skills of the product managers in prioritizing budgets and features). Then, in practice, the debt is taken or repaid by the development teams, who are the ones developing the system. However, teams working in parallel in large projects do not usually have the overall picture of the system. Such overview is needed to assess if the architecture is optimal across the implementation of many teams, or if the interest of the debt that is accumulated in one part of the system is going to be paid by some teams or by the overall organization, or else what budget can be dedicated to repay a specific debt. This is why there is a need of a spanning activity such as ATD Management, which involves not only teams, but also architects and product managers.

We have dedicated several studies to develop the sub-practices for the spanning activity of ATD management. In Chapter 7 we found how accumulation of ATD is caused by several factors, internal and external to software companies. In Chapter 8 we have recognized dangerous socio-technical vicious circles often occurring in the organizations. Such phenomena lead to the accumulation of risky ATD that need to be repaid timely to avoid situations in which the interest cause development crises. The knowledge developed in Chapter 7 and 8 is necessary in order to manage (prevent, identify, prioritize and refactor) ATD and needs to be shared among the different stakeholders (teams, architects and product managers) in an appropriate way. For this reason we also evaluate this information with different stakeholders (Chapter 9).

The information found in this part of the thesis is useful in order to develop methods and tools that can be used to support the spanning activity of ATD management. However, a practice that would make use of this information, in order to be effective, needs to be introduced in the companies and therefore needs to be combined with the existing organization and processes (in case of the companies studied here, related to

large Agile software development, as explained in section 2.2). In practice, the ATD information needs to be retrieved, analyzed and used by specific roles in the organization. In the last part of this thesis (Chapter 10), we developed and evaluated in practice an organizational solution that has been applicable in companies and has given several benefits to the developing organizations. The proposed organizational setting, including three different virtual teams connecting the main stakeholders (teams, architects and product managers) provides the expected mitigation of *interaction challenges* by facilitating the spanning activity dedicated to ATD management.

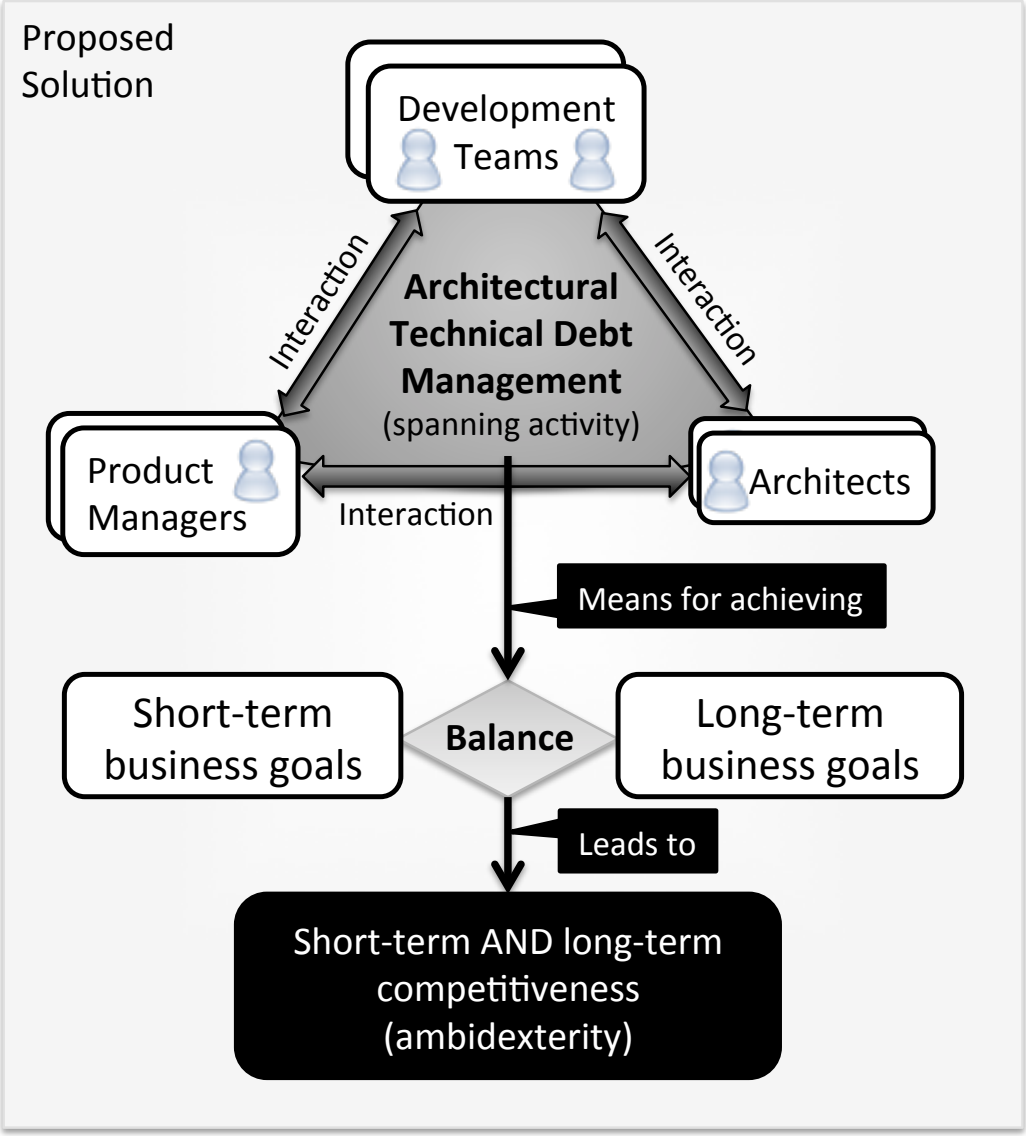


Figure 2 We have investigated a possible solution: Architectural Technical Debt management is a spanning activity that is a means to balance short-term and long-term business goals and therefore leads to achieve competitiveness in both short-term and long term, leading to ambidexterity.

In summary, the ATD management spanning activity would help the stakeholders in mitigating their existing interaction challenges causing the unbalance of short-term and long-term business goals. This practice is therefore the *means* for large software companies to be competitive both on short and long term, which would make them more ambidextrous.

1.2.3 Contributions of this thesis

The main contributions of this thesis are the followings:

- Novel scientific knowledge about what creates the unbalance between short-term and long-term business goals in large software companies:
    - We compiled a comprehensive taxonomy of the main factors affecting the balance of short-term and long-term business goals (respectively called *speed* and *reuse* in Chapter 4).
    - We identified and quantified the main interaction challenges between development teams and the rest of large software organizations. These challenges prevent the achievement balance between short-term and long-term business goals (Chapters 5 and 6).
    - We explain how ATD accumulation unbalances the organization towards short-term rather than long-term business goals (Chapters 7 and 8).
- Novel scientific knowledge of the ATD phenomenon. This includes ATD-related information to be used for methods and tools dedicated to the spanning activity of ATD management:
    - A taxonomy of the causes of ATD accumulation, useful for the avoidance of known pitfalls
    - Two qualitative models: the phases model on the trends of ATD accumulation over time in different development phases and the crises points model, useful for applying refactoring strategies and avoiding reduced responsiveness
    - A taxonomy of the most expensive (in terms of interest) ATD issues found at the studied companies, important for ATD identification and prioritization
    - Indicators for recognizing the previously mentioned ATD issues
    - Models of socio-technical anti-patterns such as Contagious Debt and other vicious circles generating ATD, useful for preventing ATD to spread in the system
- A comprehensive framework, CAFFEA, for employing the ATD management practices into a software company according to its Agile organization and process:
    - We developed and evaluated in practice an organizational solution at the studied companies, supporting the spanning activity of ATD management among teams, architects and product managers.

2 BACKGROUND

In the previous section we have presented the overall research framework of this thesis. In the following sections, we explain the various theoretical and empirical background elements that are the backbone of this thesis: we also introduce the specific Research Questions and their connection to the contributions in the rest of the chapters.

First we will take a business perspective, by defining ambidexterity in general and, as it has been considered in this thesis, as the successful balance of short-term and long-term business goals. Then we will take an organizational and process perspectives, describing the context of Large Scale Agile Software Engineering in combination with embedded software development and explaining how we mitigate, in this thesis, existing interaction challenges related to such context. Then we will explain how architecture is a key aspect that needs to be taken in consideration in order to achieve ambidexterity and how managing Architectural Technical Debt can be considered a solution for improving the mentioned challenges. Finally, we will describe how we have used existing theoretical frameworks on coordination theories and organizational boundaries to study interactions and coordination.

2.1 BUSINESS PERSPECTIVE: AMBIDEXTERITY

2.1.1 Definition of ambidexterity

As introduced in Section 1.2, the main research problem is based on understanding how to balance short-term and long-term business goals in order to sustain continuous competitiveness. This ability is called *ambidexterity*, and in this section we define ambidexterity and we relate it to the research problem and the first research question that we answered in this thesis.

The term ambidexterity can be intended as generically balancing between two conflicting goals, for example efficiency and flexibility [1]. In the specific domain of software development [2], ambidexterity is more often referred as the balance of conflicts such as the "*need to emphasize repeat-ability of development processes on the one hand and response-ability to dynamic market conditions on the other*" [3] and "*make sure that the product and project portfolios satisfy existing customers while also allowing for market expansion*" [4].

The two quotes reported above refer to two different dimensions: the first one is about a conflict in the present (internal vs external focus), while the second is about focusing on short-term goals vs long-term goals. In this thesis, we take in consideration a third perspective, which is the combination of such dimensions: we consider the response-ability with respect to existing customers (short-term) and future market demands (long-term). This means focusing on what is also regarded as *responsiveness: "Responsiveness is the ability to react purposefully and within an appropriate time-scale to customer demand or changes in the marketplace, to bring about or maintain competitive advantage"* [5]. Responsiveness is considered an important business goal of a company, since "*provid[ing] the right product within an acceptable timeframe is an essential cornerstone of sustained competitiveness"* [5]. Achieving responsiveness in short-term and in long-term are therefore the two conflicting business goals that need to be balanced.

In practice, the outcome of software development is a product (for example, a car and its software) or a service (for example, a website), which is then used by one or more customers according to their needs. During the last decade, software companies have tried to shorten the time between the identification of their customers' needs and

the delivery of a software solution that would satisfy such needs [6], which is to say they have tried to increase their responsiveness. With respect to responsiveness, software companies are considered ambidextrous if are *"aligned and efficient in their management of today's business demands, while also adaptive enough to changes in the environment that they will still be around tomorrow"* [7].

It's therefore important, in order to sustain competitiveness, for software companies to implement processes and organizations that support responsiveness. However, being ambidextrous with respect to responsiveness means not only being responsive in the short-term, but also keeping it stable in the long-term in order to *continuously* remain competitive, as mentioned in Section 1.2. In this thesis, we consider ambidexterity as the simultaneous achievement of these conflicting business goals, short-term and long-term responsiveness.

A number of approaches have been suggested in order to achieve ambidexterity: for example, *structural ambidexterity* [7] relies on the existence of two different organizational units dedicated to the two different goals, in practice separating the different conflicting concerns. However, such an approach has been found problematic in software companies [2]. In contrast, it has been proposed a different approach, such as *contextual ambidexterity* [2]. This approach suggests the achievement of ambidexterity by not influencing and relying on high-level manager only, but rather involving all individuals in the software development process (software developers, system and software architects, product managers, and other stakeholders), in achieving ambidexterity. In this thesis, when referring to ambidexterity we refer to contextual ambidexterity, since this has been recognized as more suitable for software development.

Although the concept of ambidexterity has been proposed in literature, the scientific community is still lacking a deep understanding of ambidexterity [8], with contrasting experiences reported on different success factors and different solutions. Specifically for software development, although a first attempt has been made in [2] to apply the concept of ambidexterity by extracting action principles from a single case-study on a small software company, it has proven challenging to find experiences and software engineering practices empirically evaluated in large software companies.

To summarize, in order to be ambidextrous, a software organization needs to reconcile the conflicts between short-term and long-term responsiveness in order to make both goals achievable to some degree [8]. Reconciling this conflict means balancing different factors: for example, allocating resources and creating a development environment that would promote both the goals and at the same time would not hindering them. However, such factors need first to be recognized. Therefore, the first step that we have done in this thesis is to answer the following RQ:

RQ1 What factors influence ambidexterity?

A first exploratory and holistic answer to this RQ is investigated in Chapter 4. As mentioned in section 1.2, a high-level factor influencing the balance of short- and long-term business goals (and therefore ambidexterity) is the conflict between Agile practices and reuse (the connection is explained in the next section). In Chapter 4 we investigate this relationship, although we don't refer directly to Agile but to its claimed benefit, speed (in the short-term).

2.1.2 Ambidexterity in Software Engineering

There have been several approaches to manage software development since the 50ies. However, a first approach for increasing (long-term) responsiveness has been proposed about 15 years ago and is the Software Product Line Engineering (SPLE) approach. Literature reports how *"some cases of SPLE have reported improvements in*

the order-of-magnitude with respect to cost, quality, and time-to-market" [9]. Its strengths rely in a preliminary phase called *Domain Engineering*, which is responsible for creating a platform able to optimize the *reuse* of software (components) by anticipating future *variability* among the products that are developed by the software companies. This way, SPLE focuses on improving long-term responsiveness by investing initial resources in order to plan and prepare a platform that would support reusability and therefore the quick replication of products based on the first product delivered. As explained in 1.2, reuse is one of the main means to achieve long-term responsiveness. However, the SPLE approach suffers from slow release cycles [10], which hinder the competitiveness given by short-term responsiveness to changing requirements [5]. Therefore, the SPLE approach tends to unbalance software development towards long-term business goals rather than short-term ones. This is in conflict with the definition of ambidexterity, so SPLE cannot be considered (alone) as a means to achieve ambidexterity.

In order to increase short-term responsiveness, in the last decade the software engineering community has seen the establishment of new software development paradigms, such as Agile Software Development (ASD, see next section) [6][11]. ASD is directly connected to short-term responsiveness, as *"At its core, agility entails ability to rapidly and flexibly create and respond to change in the business and technical domains"* [6]. ASD is based on the concept of iterative development, where the development does not rely on an extensive preliminary phase in charge of predicting and taking care of all the possible future requirements, but it's more dedicated to reacting upon changing requirements. Among other aspects, ASD is focused in improving short-term responsiveness by increasing the deliveries throughout the development process, in order to quicken the feedback from the customer and to adjust to changing needs (requirements).

Although ASD aims at improving short-term responsiveness, it's still not clear what are the impacts of such an approach on long-term responsiveness. In this thesis we investigate the challenges in large companies employing ASD, in order to understand what factors are hindering the achievement of long-term responsiveness, necessary to be balanced with the short-term one in order to achieve ambidexterity.

2.2 Organization and Process Perspective: Large-Scale Agile Software Development

2.2.1 Large-Scale Agile Software Development

Agile and Lean Software Development have been associated, among other given benefits, with short-term responsiveness, as explained in the previous section. ASD has been extensively studied in the last years [6] since 2001, when the Agile Manifesto was created [12]. A number of methods have been developed, but very few empirical evidences were available on their actual benefits if not on a principle level [13]. A literature review in 2008 [14] highlighted how the most studied method was XP [15], which was however reported to rarely be completely applicable in practice and only suitable for small companies but not for large projects. Subsequently, the focus shifted also to large projects [16], in which the Scrum method became more studied because it seemed better adaptable for industry, as mentioned in [6]. A trend in the last few years has been to combine ASD with Lean Software Development (LSD) [6], [17], [18]. LSD is another development paradigm aimed at improving efficiency and eliminating waste throughout the whole product development from a holistic point of view.

2.2.2 *Large-Scale Embedded Software Development and Agile*

Embedded software is developed to control machines or devices that are not considered generic purpose computers. Such software is typically specialized for the particular hardware that it runs on and has to match specific time and memory constraints. Examples from such domains are cars, telephones, modems, robots, appliances, toys, security systems, pacemakers, televisions and set-top boxes, and digital watches.

As mentioned in the previous section, ASD have been the subject of researchers' attention in the last years [6]. Companies developing software for specific domains, such as embedded software development, have shown the trend to adopt ASD [19],[20]. However, such development process shows properties and difficulties that are not common to generic software development, and have to be taken care of when employing ASD. Examples are the necessity for the Agile teams to depend on strict hardware requirements and resources and to the necessity of following the overall product development [19],[20],[21].

2.2.3 *Agile teams and large organizations*

In small companies or small projects, an Agile development team might be alone working on a project or might be connected with only a few other entities internal or external to the company [22]. However, in large projects, the Agile development team has to interface with several other teams and with other (social) groups [16]. In Chapter 6 we have investigated what are these other social groups.

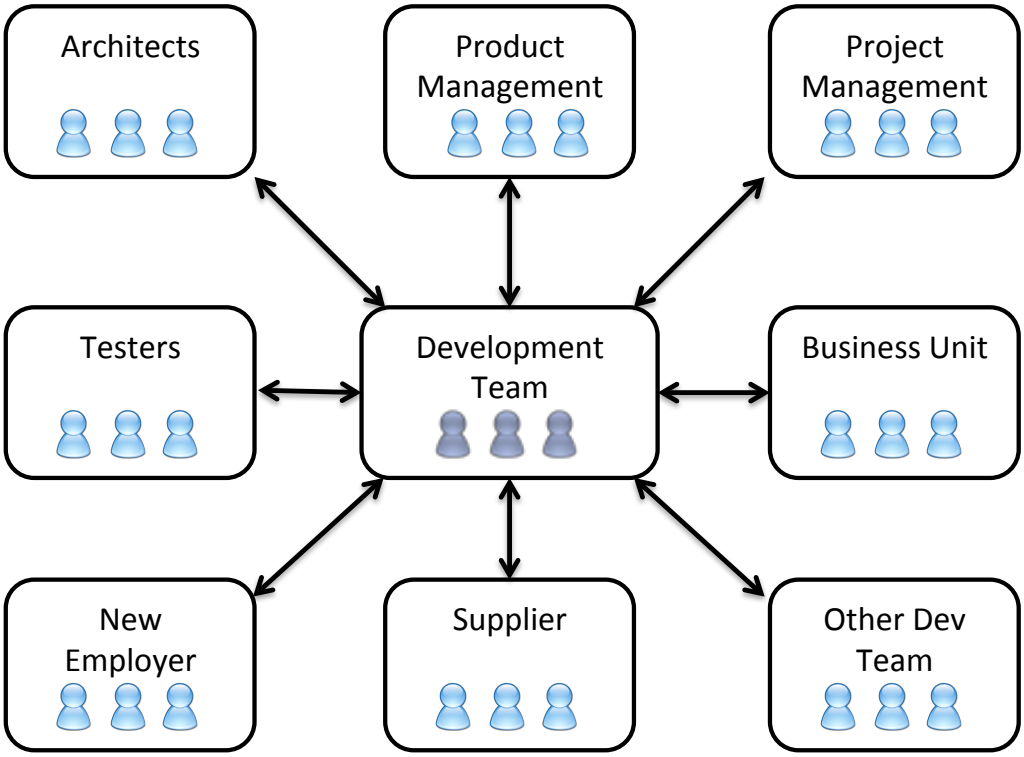


Figure 3 Interaction connections between the Agile Development Team and the rest of the organization.

The diagram in Figure 3, extracted from Chapter 6, shows many possible connections, especially rich for embedded software development, where not only software but also a physical product is developed (e.g. a car). Such large number of communication links increases the complexity of software development process [23]. Given the many dependencies among the different groups involved, it becomes very complex to coordinate short-term and long-term goals across large projects. This problem is still not well understood in ASD [6], and a recent study [24] reported on the

difficulties of aligning the Agile teams with respect to short-term and long-term goals, as there is *"often a conflict between the need for short-term progress and the need for long-term product quality at the end of sprints"*. Although another study shows how an Agile team can achieve ambidexterity internally [25], it remains an open issue how to achieve it among different teams and other social groups.

In summary, aligning different parts of large Agile organizations developing embedded software in order to achieve ambidexterity remains an open issue, and this in turn does not allow large companies to control the balance of the two goals. This challenge motivated the second research question:

RQ2 What interaction challenges affect ambidexterity?

In this regards, Chapters 5 and 6 have been dedicated to understand what challenges are hindering team interactions with other teams and with other parts of large Agile organizations, with respect to short-term and long-term responsiveness. We explain, in the next section, the context in more details in order to introduce the next research questions.

2.2.4 Interaction challenges when prioritizing features and architecture in Agile Software Development

Short-term responsiveness is usually achieved, in ASD, by prioritizing the most important features in the beginning of the development. Such activity is ideally carried out by the customer, but in practice, especially in large organizations, this task is performed by a surrogate of it, called *Product Owner* (PO) [26]. The PO usually prioritizes a *backlog* for the team, which then invests a certain amount of time, called sprint (usually a short cycle, such as few weeks), in order to develop the features appearing on the top of such list. By continuously re-prioritizing the backlogs each sprint, according to changing requirements [27], and by performing continuous integration [28], in which every short amount of time the various parts of the software are built and tested together, the companies have a complete product (including the most important features) that is often deliverable to the customer and includes the most important features.

Although architecture has traditionally been considered central for driving software development in the last 35 years [29], there are reports that highlight how focusing on long-term qualities is not as important as the fast and short-term delivery of features. Also empirical evidence suggest that *"Agile methods often tend to de-prioritize architecture, leaving architecture erosion as a consequence."* [10]. This is a clear sign that long-term goals are unbalanced in favor of short-term goals. This issue is due to three main interaction challenges (summarized here and explained more in details in Chapters 5 and 6), better understandable if we take a closer look at the Agile prioritization process.

Figure 4 represents the overall prioritization process in balancing short-term and long-term responsiveness. *Business goals* are used in *Prioritization,* in order to drive the *Development* activity. Also the *Development* outcome is *Assessed* using the business goals. The result of the *Prioritization* activity is a plan of action for the *Development*. The outcome of the *Development* is then used in the *Assessment* activity of the software with respect to the business goals. The Assessment provides feedback both to change plans via further *Prioritization* (for example, if the business goals are not met), or directly by the teams during the *Development*. Figure 5, Figure 6, Figure 7 and Figure 8 describes more in details how each activity in related to the others.

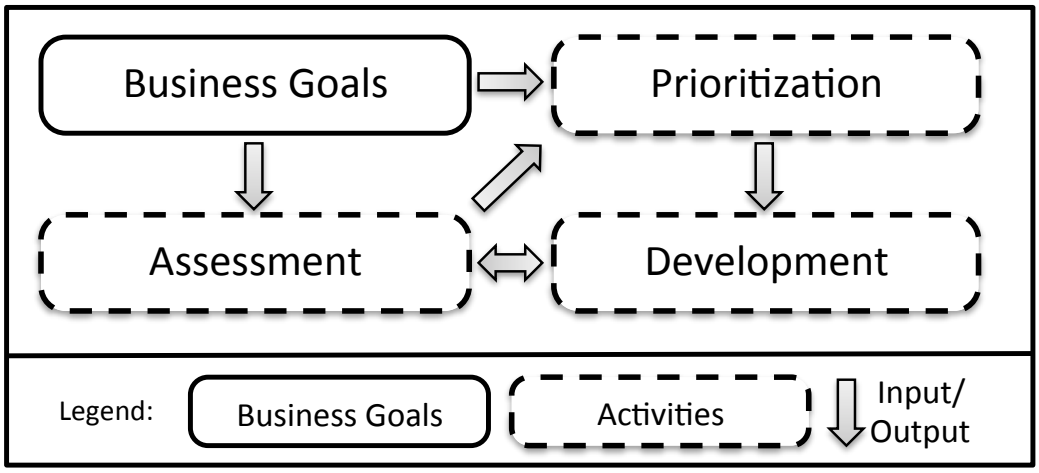


Figure 4. The overall process, including several activities and input/output among them: Business Goals are used both for Prioritization and Assessment. The outcome of Prioritization is used during Development, while the outcome of the Development is used for Assessment. This activity provides feedback for both Prioritization and Development activities.

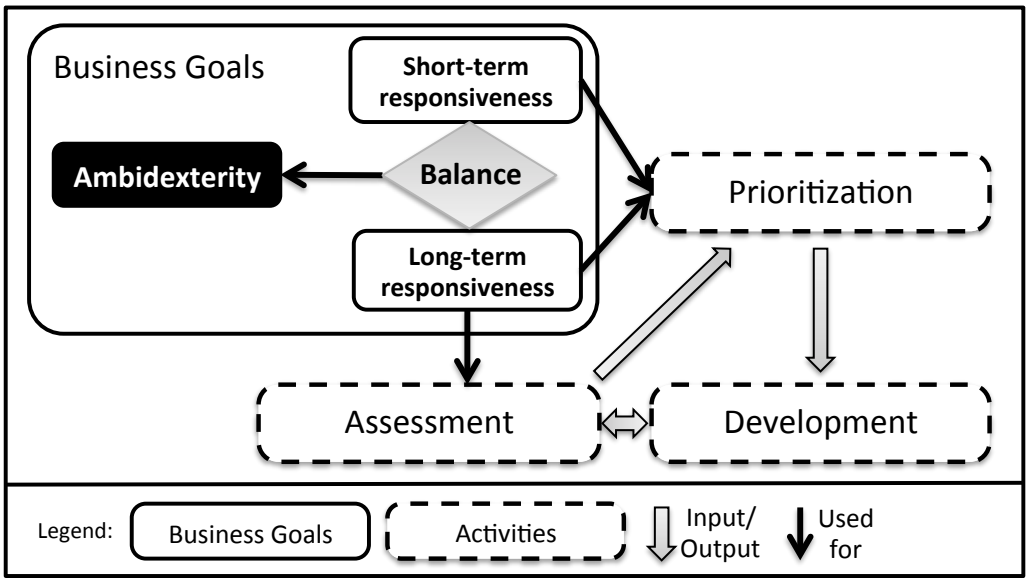


Figure 5 The Business Goals analyzed in this thesis are short- and long-term responsiveness. Their balance makes companies ambidextrous. Both business goals are used in Prioritization. Long-term responsiveness is also used in the Assesment of the output of the software Development.

The balance of two business goals, short- and long-term responsiveness leads to ambidexterity (Figure 5). Such goals are used in the *Prioritization* activity as *Prioritization aspects* both for *Product Owners (POs)*, and *Architects (Archs)*, The outcome of this prioritization is communicated to the Agile teams (*Development*).

The first interaction challenge is the *Alignment* (Figure 6) between these two roles and their views. The outcome of such activity is a plan of action, including respectively a prioritized list of *Feature requirements* from the POs and a specification of the *Architecture Significant Requirements* from the *Architects* to several *Development Teams*. As shown in Figure 7 each development team receives both kinds of input and then produces a *Technical Solution* (it might be a feature or an *increment* in general). However, these two inputs for the *Development* might be in conflict between each other, as explained in Chapter 6. This is the first source of unbalanced goals.

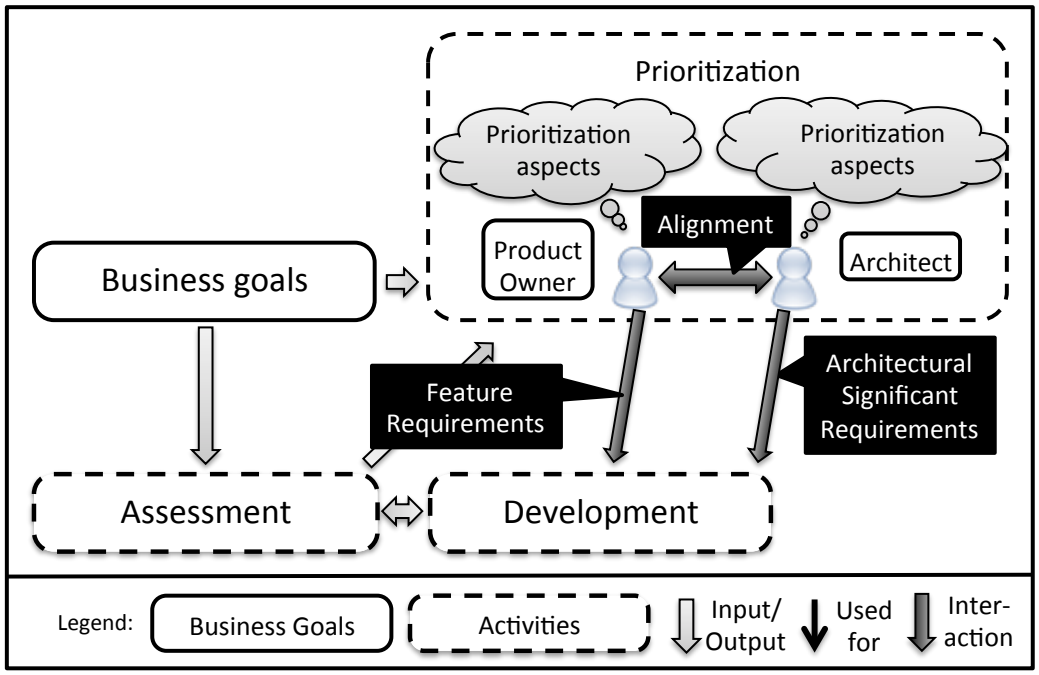


Figure 6 Architects and product owners prioritize based on prioritization aspects and communicate their (aligned) input, Architectural Significant Requirements or Feature Requirements, to the Development teams.

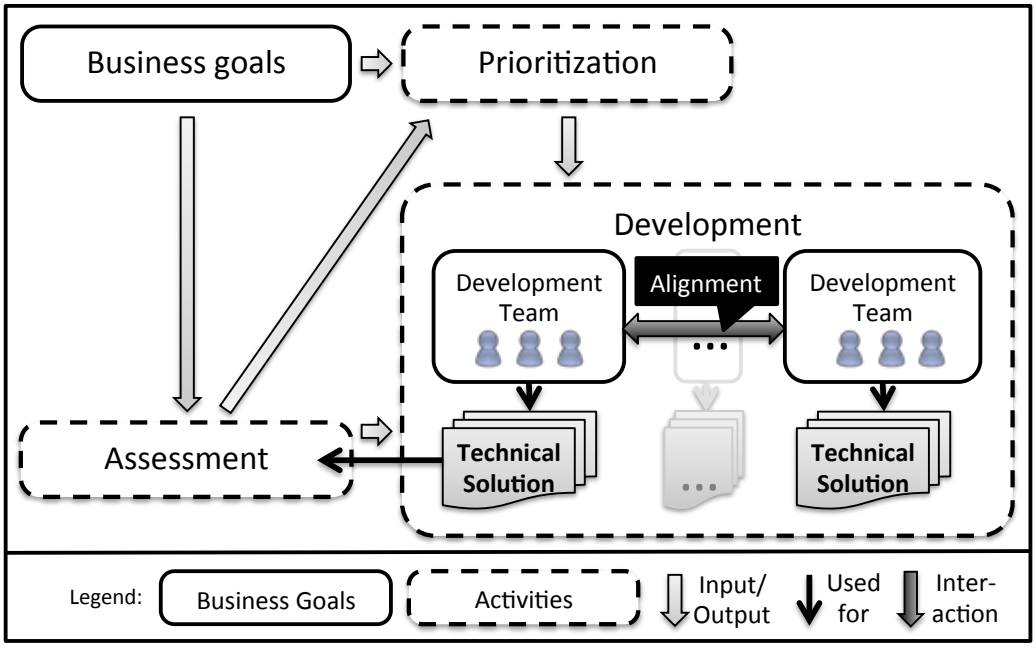


Figure 7 The teams (should) align with each other and then develop a technical solution to be delivered to the customers. However, the software can be Assessed in order to understand if it satisfies the long-term qualities.

The second challenge is also related to *Alignment*, but among the teams (described in Chapter 5). The developed *Technical Solutions* might not equally satisfy the quality requirements necessary for long-term responsiveness. For example, one feature might be developed according to the architecture, but another one might create several dependencies that are not allowed (creation of Architectural Technical Debt).

The third challenge is related to the need for an *Assessment* of the actual technical solution with respect to long-term responsiveness (Figure 8). It is possible to understand if the process supports achieving short-term responsiveness thanks to the

continuous integration process: if the software is built and released to the customer and the customer is satisfied, then the solution clearly supports short-term responsiveness and it's possible to assess such quality iteratively. However, something that is more difficult to assess is if the system is complying with the *Qualities* defined in the architecture in order to achieve long-term responsiveness. There is currently a lack of suitable mechanisms, for example an *Architectural Technical Debt map*, which would show if such quality is achieved. This assessment would also make visible the *Risk* that the solutions will not be able to support long-term responsiveness. The lack of such assessment, and therefore of *Feedback* to POs, Architects and Development Teams, makes it difficult for such actors to act upon such critical information and therefore to adjust their activities in order to balance short- and long-term responsiveness. The outcome might be that the balance between the two business goals is not achieved and the company either focuses too much in only one of the goals, or it remains stuck in the decision process between the two goals because of conflicts raising among the actors, which in turn causes performance penalties.

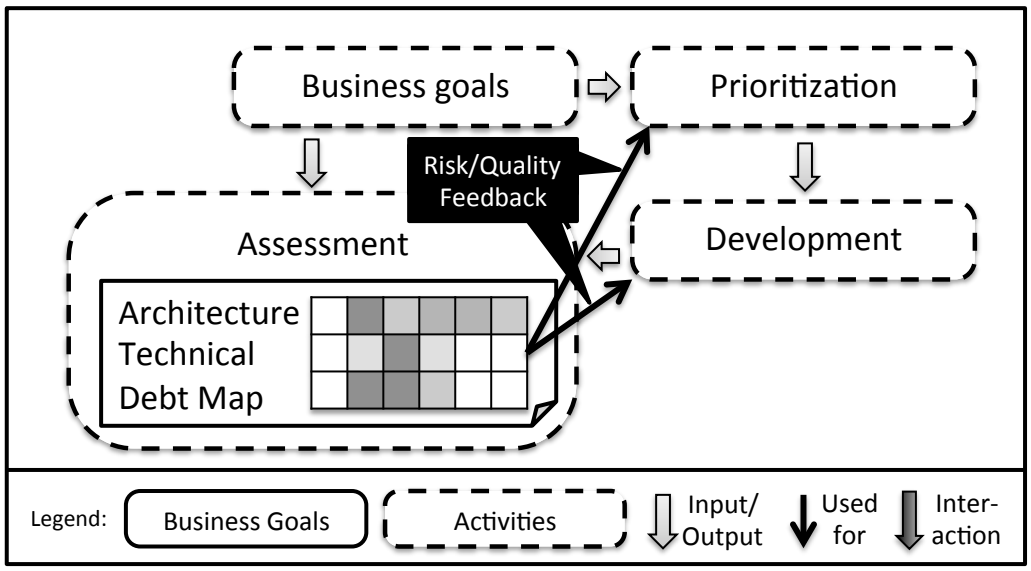


Figure 8 The developed software can be Assessed in order to understand if it satisfies long-term qualities: otherwise, there is the risk that the solution would not be able to support long-term responsiveness. Such assessment gives feedback to the stakeholders in the Prioritization and Development activity, who can reiterate the process in order to improve the balance.

In order to overcome these three interaction challenges involving several product managers, architects and teams in large-scaled Agile Software Development, there is a need for developing suitable *spanning activities* that would aid the interaction of such actors. These activities are supposed to be conducted by the different actors involved, in order to mitigate the challenges (spanning activities are explained with more details in section 2.4, when we explain the *coordination* theoretical framework used for investigating a solution). The third research question is therefore:

RQ3 What spanning activities are needed in order to mitigate the interaction challenges affecting ambidexterity?

We contribute to this research question with the recommendations for inter-team interaction in Chapter 5, while in Chapter 6 we list a number of spanning activities that are needed among several groups in large companies developing embedded software (among which we propose the Architectural Technical Debt management activity).

2.3 ARCHITECTURE PERSPECTIVE: ARCHITECTURAL TECHNICAL DEBT

2.3.1 Software Architecture

Long-term responsiveness is usually achieved by providing a technical solution of the system that would satisfy specific qualities, anticipating long-term goals (for example reusability, flexibility, etc.). The achievement of such qualities is usually based on an *architecture* defined by software experts (also called *Architects*): the architecture of a system captures the desired properties of the system with the specification of significant non-functional requirements to be satisfied or the specification of patterns and rules to be followed when the features are developed. In other words, architecture is defined as *"the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution"* [30].

2.3.2 Technical Debt and Architectural Technical Debt

As mentioned in the previous section, there is a need for a mechanism that would create information to be assessed by architects, managers and development teams in order to assess the quality of the software with respect to long-term responsiveness, and therefore to be aware of the risks of not being able to sustain such responsiveness. A suitable mechanism for making such information available is related to the *Technical Debt* (TD) concept. This metaphor is borrowed from the financial domain and it relates implementing sub-optimal solutions, developed to meet short-term goals, to taking a financial debt, which has to be repaid with interests in the long-term. Such metaphor has been recently researched in academia and in industry, as shown by a recent systematic mapping study on the subject [31].

The term Technical Debt has been first coined at OOPSLA by Cunningham [32] to describe a situation in which developers take decisions that bring short-term benefits but cause long-term detriment of the software. The term has recently been further studied and elaborated in research: in 2013 Tom et al. [33] conducted an exploratory study involving a multi-vocal literature review, supplemented by interviews, in order to draw a first categorization of TD and the principal causes and effects.

Some studies have been conducted on the management of TD, also supported by a dedicated workshop (MTD), and usually co-located with premium conferences, such as ICSE and ICSME. A first roadmap has been created in 2010 by Brown et al. [34]. In 2011 Guo et al. proposed an initial portfolio approach with the creation of TD *items* to be managed in the companies. The same authors proposed a further empirical study on tracking TD [35], in order to make the lack of quality and the risk visible for a software development team. In relation to the problem of neglecting long-term goals in favor to the short-term ones, Seaman et al. have identified the theoretical importance of TD as risk assessment tool in decision making [36]. TD has also been used for defining part of a method for assessing software quality, SQALE [37], which has also been implemented in a tool for source code static analysis. The Technical Debt items can be used in practice as explained in Figure 9.

A more specific kind of Technical Debt is regarded as Architectural Technical Debt (ATD, categorized together with Design Debt in [33]). A further classification can be found in Kruchten et al. [38], where ATD is regarded as the most challenging TD to be uncovered since there is a lack of research and tool support in practice. Finally, ATD has been further recognized in a recent systematic mapping [31] on TD.

Analogously to TD, Architectural Technical Debt (ATD) is regarded [33] as *sub-optimal solutions* with respect to an optimal architecture for supporting the business goals of the organization. Specifically, we refer to the architecture identified by the software and system architects as the optimal trade-off when considering the concerns

collected from the different stakeholders. In the rest of the thesis, we call the sub-optimal solutions *inconsistencies* between the implementation and the architecture, or *violations*, when the optimal architecture is precisely expressed by rules (for example for dependencies among specific components). However, it's important to notice that (in the cases studied in this thesis) such optimal trade-off might change over time, as explained in Chapter 7, due to business evolution and to information collected from implementation details. Therefore, it's not correct to assume that the sub-optimal solutions can be identified and managed from the beginning.

2.3.3 Architectural Technical Debt theoretical framework

The studies in TD and ATD are quite recent, and the subject is not mature yet. Some models, empirical [39] or theoretical [40] have been proposed in order to map the metaphor to concrete entities in software development. We use, in this thesis, a conceptual model comprehending the main generic components of TD.

2.3.3.1 Debt

The debt is regarded as the technical issue. Related to the ATD in particular, we consider the ATD item as a specific instance of the implementation that is sub-optimal with respect to the intended architecture to fulfill the business goals. For example, referring to Chapter 8, a possible ATD item is a dependency between components that is not allowed by the architectural description or principles defined by the architects. Such dependency might be considered sub-optimal with respect to the *modularity* quality attribute [41], which in turn might be important for the business when a component needs to be replaced in order to allow the development of new features.

2.3.3.2 Principal

It's considered the cost for refactoring the specific TD item. In the example case explained before, in which an architectural dependency violation is present in the implementation, the principal is the cost for reworking the source code in order to have the dependency removed and the components not being dependent from each other.

2.3.3.3 Interest

A sub-optimal architectural solution (ATD) might cause several effects (for example, as described in [42]), which have an impact on the system, on the development process or even on the customer. For example, having a large number of dependencies between a large amount of components might lead to a big testing effort (which might represent only a part of the whole interest in this case) due to the spread of changes. Such effect might be paid when the features delivered are delayed because of the extra time involved during continuous integration. In this paper, we treat accumulation and refactoring of ATD as including both the principal and the interest.

2.3.3.4 The time perspective

The concept of TD is strongly related to time. Contrarily to having absolute quality models, the TD theoretical framework instantiates a relationship between the cost and the impact of a single sub-optimal solution. In particular, the metaphor stresses the short-term gain given by a sub-optimal solution against the long-term one considered optimal. Time wise, the TD metaphor is considered useful for estimating if a technical solution is actually sub-optimal or might be optimal from the business point of view. Such risk management practice is also very important in the everyday work of software architects, as mentioned in Kruchten [43] and Martini et al. [44]. Although research has been done on how to take decision on architecture development (such as ATAM, ALMA, etc. [45]), there is no empirical research about how sub-optimal architectural

solutions (ATD) are accumulated over time and how they can be continuously managed.

2.3.4 *ATD as spanning activity to achieve ambidexterity*

The Technical Debt metaphor seems to be a suitable communication device among the stakeholders mentioned in 2.2 for aligning short-term and long-term goals. In particular, part of the overall TD is to be related to architecture sub-optimal solutions, and it's regarded as Architecture Technical Debt (ADT). ATD is considered as implemented solutions that are sub-optimal with respect to the quality attributes (internal or external) defined in the desired architecture intended to meet the companies' business goals. Although there are very few scientific reports on the benefits of employing ATD in practice, several experiences from the author of this thesis have shown how the recent introduction of such metaphor in the industrial partners of this research project have contributed in improving the communication among the different stakeholders.

The following Figure 9 shows how Architectural Technical Debt can be used in the previously presented framework (Figure 4). The information about *Quality* and the consequent *Risk* in terms of threats for long-term responsiveness is *Assessed* in the Technical Solutions developed by the Development Teams. As suggested in previous research ([46]), and effective way of visualizing and utilizing the Technical Debt information is to create a Portfolio (or Backlog) with TD items. We use the same idea: including the information about ATD in an Architecture Technical Debt Map. Such map is then shared between Product Owners, Architects and Development Teams. Such information will therefore contribute to the *Alignment* among the stakeholder (especially between Product Owners and Architects but also among Development Teams). They need to take in consideration the risk of unbalancing the development in favor to short-term responsiveness by neglecting qualities necessary for achieving long-term responsiveness.

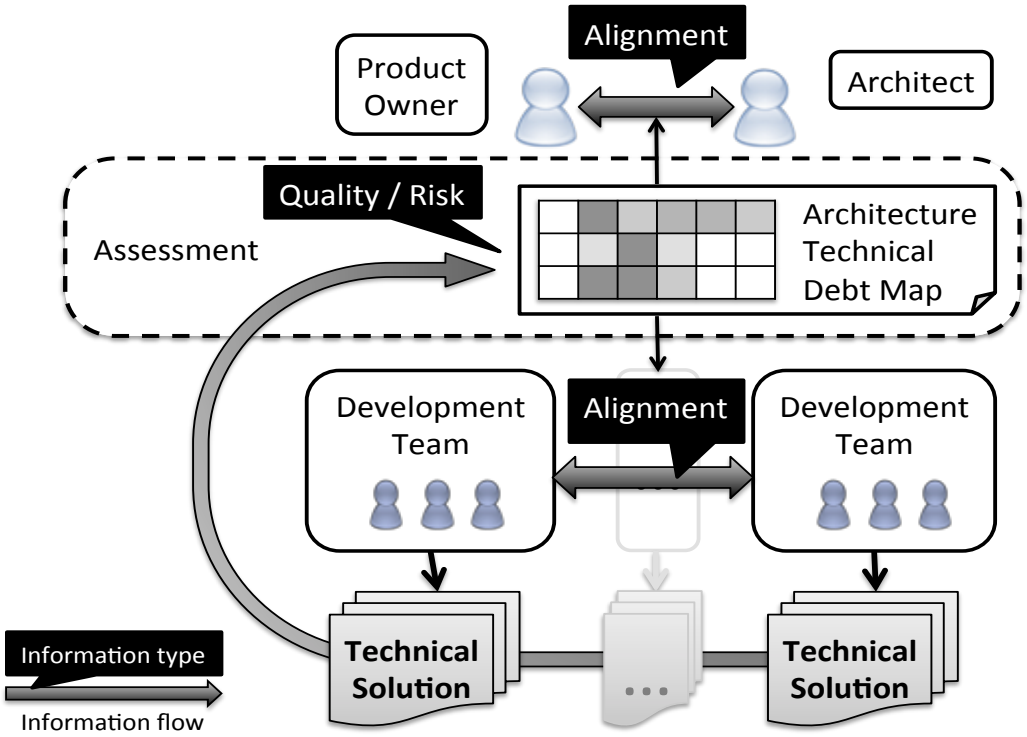


Figure 9. Example of a spanning activity in which Development Teams, Architects and Product Owners align and assess short-term and long-term responsiveness using an Architecture Technical Debt Map.

In order to use ATD management as a suitable spanning activity, it's important to understand what information needs to be shared in such spanning activity, i.e. what interactions need to be improved. Therefore, we want to answer the following research question:

RQ4 What strategic information about Architecture Technical Debt needs to be shared between architects, product owners and teams in order to manage ambidexterity?

Although ATD management has been recognized as usable in practice, a recent comprehensive study of existing literature [31] highlights several deficiencies in the current body of knowledge: lack of reliable industrial studies, lack of focus on architecture anti-patterns and lack of studies involving the whole TD management process. This means that, although the idea of using TD management has been already formulated, the implementation of such mechanism, including tools and its integration in the software processes, is far from being fully employed in practice, as also highlighted in Chapter 10.

We contribute to filling this gap with three in-depth studies of ATD, presented in the second part of this thesis, including Chapters 7 and 8, where we have studied the phenomenon in practice in order to build knowledge and useful information, which we have started to evaluate with the involved stakeholders in Chapter 9.

2.4 Interaction and Coordination Perspective

In the previous sections of this Chapter, we have presented the background elements underlying the Research Questions (RQs) and contributions of this thesis. We have used, as a structure, the BAPO theoretical framework [47] (Business, Architecture, Process and Organization, not necessarily in this order): such framework is extensively studied and referred in recent research (e.g. in [48]) and successfully applied in practice [47][49], both for the understanding and the assessment of various aspects of software development.

However, as explained throughout the section several factors influencing the balance of short-term and long-term responsiveness were related to the *Interactions* (interaction challenges) among the involved stakeholders. This aspect is orthogonal to the previously mentioned categories (BAPO), being related to the *People* (or social) aspect of software development, which has also been recognized in other frameworks, such as in [50]. The *interaction* dimension is also considered one of the pillars of the four main ASD principles, as outlined in the first line of the Agile manifesto [12], where the founders of ASD mention how there is more value in *"Individuals and interactions over processes and tools"* (even though processes and tools have value as well). In order to investigate the interaction challenges related to ambidexterity and in order to develop a suitable solution (ATD management as spanning activity), we therefore needed theoretical frameworks to underpin this aspect of our research.

In this section we introduce the concepts and terminologies that are extracted from two theoretical frameworks, respectively from the theories on communication and coordination. These concepts are useful to understand the theoretical underpinning and the contributions of this thesis with respect to *interactions*.

2.4.1 Internal and external interactions

According to the theory of organization, between two social groups (for example, in software development, these can be teams of different kinds, development, business, architecture, cross-functional, etc.) stands an *organizational boundary*, defined as *"the demarcation of the social structure that constitutes an organization"* [51]. As an example, when splitting and assigning developers to two Agile teams, automatically an

organizational boundary is created between the developers belonging to the two teams. This has several implications: we do not discussed all of them here, but we pay particular attention to how organizational boundaries affect the interaction between different groups. In fact, the interaction between two groups crosses the organizational boundaries that stands between them, and is defined as *communication*. This is regarded as the activity of conveying information through the exchange of thoughts, messages or information, as by speech, visuals, signals, writing or behavior. Although teams are created to increase the *internal* communication among a few employees, the creation of organizational boundaries among a large number of groups might create a decrement of the *external* communication among these groups, since boundaries function as barriers. Finally, interaction across boundaries is more expensive.

2.4.2 Coordination strategy to manage interactions with spanning activities

To overcome the barriers created by the organizational boundaries, special communication practices have the function of facilitating the communication across such barriers, or in a figurative way they have the effect of *spanning* across the boundary (for example, they have the same effect that a bridge has on a river). For this reason, such communication practices have been recently referred as *spanning activities* in information system research [52]: "*Boundary spanning occurs when someone within the project must interact with other organizations, or other business units, outside of the project to achieve project goals. There are three aspects to boundary spanning: boundary spanning activities, the production of boundary spanning artifacts, and coordinator roles*". Such concept has been used with respect to Agile Software Development by Strode et al. [22]. Spanning activities are seen as part of an overall coordination theory that includes roles, object and spanning activities. This framework is based on the Malone et Crowston's coordination theory [53], and Strode et al. analyzed how co-located and small agile projects coordinate and what spanning activities they used. We have used such framework in Chapter 6, where the same theoretical framework has been used and the results have been presented in the form of needed spanning activities between the Agile team and the other parts of the organizations. Strode's framework proposes the definition and description of *spanning activities*, of *spanning objects* (artifacts) and the *coordinators* (roles). Each activity needs to be carried out also at a specific frequency, i.e. *per-project*, *per-iteration* and *ad-hoc*. According to the previous definition and the findings highlighted by Strode et al. [22], Levina and Vaast [52] and Malone and Crowston [53], it's necessary to establish spanning activities for mitigating interaction challenges. This is why we study, in this thesis, not spanning activities in general, but a specific instantiation of a spanning activity dedicated to the interaction challenge at hand, as explained below.

Given the research problem explained in 1.2 and the interaction challenge outlined in 2.2.4, we came to the conclusion (Chapter 6) that Architectural Technical Debt management would be a suitable spanning activity among product managers, architects and development teams, as shown in Figure 9. The second part of this thesis is therefore dedicated to the understanding and creation of ATD-related strategic information that needs to be conveyed (communicated) among the interacting stakeholders in the spanning activity.

However, this spanning activity, in order to be effective, needs to be introduced in practice in large companies and therefore needs to be combined with a solution that would fit the roles in the organizations and processes (in the studied context represented by the large-scale ASD). This leads to the fifth RQ:

RQ5 What organizational solution can be applied in order to facilitate spanning activities to manage ambidexterity?

The contribution to this RQ is given in Chapter 10: we propose an organizational solution that needs to be introduced in the large Agile process and organization. The overall framework is called CAFFEA: this allows product managers, architects and Agile teams to perform the necessary spanning activity of managing Architectural Technical Debt and therefore to better balancing short-term with long-term goals and therefore allowing organization to be more ambidextrous.

2.5 MAPPING OF RESEARCH QUESTIONS AND CHAPTERS

The table below summarizes the research questions and their link to the following Chapters of this thesis.

Table 1 Map of Research Questions and Chapters

RQ	***Chapter***
RQ1 What factors influence ambidexterity?	Chapter 4
RQ2 What interaction challenges affect ambidexterity?	Chapter 5, 6
RQ3 What spanning activities are needed in order to mitigate the interaction challenges affecting ambidexterity?	Chapter 5, 6
RQ4 What strategic information about Architecture Technical Debt needs to be shared between architects, product owners and teams in order to manage ambidexterity?	Chapters 7, 8, 9
RQ5 What organizational solution can be applied in order to facilitate spanning activities to manage ambidexterity?	Chapter 10

3 RESEARCH DESIGN AND METHODOLOGY

This Chapter describes and discusses the choices made during this thesis with respect to the research methodologies applicable in the Software Engineering field. First we will describe the research context and the collaboration between academia and industry that led to the results presented in the rest of the thesis. Then we will introduce the overall research design with the use of Grounded Theory on the strategic level and of the case-study research at the tactical level. The last sections will go into details of the research methods used for data collection and data analysis.

3.1 RESEARCH CONTEXT

This thesis is based on a research project that was set up as a collaborative effort between Chalmers University of Technology and several companies located in Gothenburg area and the whole Scandinavian area. The main purpose was to contribute to the scientific knowledge and to improve the software development processes of the involved industrial partners. Therefore the motivation for addressing the specific research problem in this thesis is not only related to gaps found in the academic body of knowledge, but is also strongly rooted in current practical needs of several large companies developing software.

3.1.1 Software Center and Agile Research Collaboration

The project behind this thesis is part of a broader initiative called Software Center, which defines a specific format to shape the collaboration between academia and industry. Since this project and its format had a substantial impact on this PhD thesis, we thing that it deserves to be explained, analyzing pros and cons.

The research format involved an initial strategic planning phase for the research project, with long-term goals, followed by continuous adjustment to achieve short-term goals and partially re-evaluate the long-term goals. This format is in line with the action principles of Agile Collaborative Research recommended in [54]. This format was followed for a number of reasons: this research project is a follow-up of the previous research described in [54], with Ericsson being one of the main partners contributing to this project. The involved partners were however expanded with respect to [54], with the inclusion of several other companies. Nevertheless, the original action principles were followed and adjusted to the involvement of more research partners.

These action principles were based on a number of success factors related to the research activities described in [54]. For example, the large participation of several companies to this project (see next section) shows how there was *management engagement* and a fruitful *network access* for the main researcher in the project (which, in this specific case, was the author of this PhD thesis). The continuous communication, evaluation of results and further direction, formally performed every six months, gave the researchers the means (*communication ability*) to report results, which in turn gave the possibility to evaluate them or setting up activities to perform a more in-depth evaluation of the results with the industrial partners. This was also very important to keep *continuity* of the project, which is the ability of continuing on the same (or slightly different) strategic goal set up in the beginning of the project: this was critically important for the progress of this PhD thesis, since it gave the author the possibility to keep the various investigations consistent and related to an overall research goal, explained in the first Chapter of this thesis.

This continuous communication with the industrial partners and their engagement was important for other reasons as well: first, it allowed us to apply both inductive

(more exploratory) and deductive (more confirmatory) approaches, increasing the strength and therefore the reliability of the results. This pattern is observable throughout the whole thesis (Figure 10): in most publications, there is always an exploratory approach and a subsequent evaluation approach that involves confirming the results with more evidence.

The continuous communication approach allowed us to evaluate the results in order to be useful for the industrial context (or, if not, a slight change in direction was applied by the researchers): this way, the results were often *aligned with industry goal* and had *benefits for industry*. For example, the CAFFEA framework (Chapter 10) has been partially or fully employed in some of the companies and is in the process of being evaluated (a first, static evaluation is visible in Chapter 10). Current evaluation at one of the companies (unpublished results) suggests how such framework had a clear impact in improving the company's development process after having been tailored for their specific context.

Finally, the research format resulted to be *innovative*: for example, in many companies the Technical Debt metaphor was introduced, as well as the awareness and the factors behind the need for balancing short-term and long-term development goals. The meeting of this success factor was corroborated by several occasions when the author of this thesis was invited for talks and presentation at several companies (even external to the research project) or, in two occasions, his participation with presentations in public events with hundreds of participants from industry such as the Software Development Day (held each year in Gothenburg).

The research format presented also some obstacles, which have been however mitigated in accordance to [54]: for example, the *negotiation* took time and resources both from the researchers and from the industrial partners, especially in the beginning, when the project was young and both the industrial partners and we needed to know each other and we needed to align different goals (indeed, this kinds of projects involve substantial *learning* effort). However, at this time the collaboration and the negotiation has clearly improved, and the negotiation is usually carried out in a smooth way, given the knowledge and the trust that has been built up during these years of collaboration.

Another partial obstacle was the difficulty to disseminate the results and goals in an effective way within the companies. Such obstacle, to be mitigated, required the author of this thesis to improve the communication of the scientific results in a *visual* way that would also fit the industrial reality rather than mentioning unfamiliar theoretical concepts. We found that this communication ability is also quite important for the actual investigation of the studied phenomena from a research perspective: adapting the theoretical concepts to the industrial context during investigation increases the internal validity of the results, since there is less probability that the informants would misunderstand the investigation device (for example, the posed question).

3.1.2 Case companies

In the following we describe the companies that were participating in the various investigations. A detailed summary of which company has been studied in which part of the thesis and the rationale for the selection of the cases is discussed and visible in section 3.3.3.

Company A carried out part of the development 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 (projects). The internal releases were short but needed to be aligned, for integration purposes, with the stage gate release model (several months).

In company B, 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 company combine 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 (several months), but the company is in transition towards reducing the release time.

Customers of Company C 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 and some with focus on the platform by different products. 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. For this company, being a very large one, we investigated several sites located across Europe.

Company D is a manufacturer of a product line of embedded devices. The organization is divided in teams working in parallel and using SCRUM. The organization has also adopted principles of software product line engineering, such as the employment of a reference architecture. Also in this case, the hardware cycle has an influence on the software release cycle.

Company E developed field equipment responsible for several large subsystems, developed in independent projects. Implementation of agile processes has been initiated, while team composition still followed formal and standard project processes due to legal responsibilities. Projects are characterized by a long-lasting product maintained for several years, by a strict formalization of the requirements and by the need of substantial testing effort.

Company F is a company developing software for calculating optimized solutions. The software is not deployed in embedded systems. The company has employed SCRUM with teams working in parallel. The product is structured in a platform entirely developed by F and a layer of customizable assets for the customers to configure. Company F 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 systems developed in Java and Python or other languages. Some companies use a Domain Specific Language (DSL) to generate part of their source code.

All the companies have employed SCRUM, and have a (internal) release cycle based on the SCRUM recommendation. However, the embedded companies (A-E) depend on the hardware release cycles, which influences the time for the final integration before the releases. Therefore, some of the teams have internal, short releases and external releases according to the overall product development.

3.2 STRATEGIC RESEARCH DESIGN: GROUNDED THEORY

The overall research strategy is based on the Grounded Theory (GT) approach [55], [56]. Grounded Theory is a systematic approach to data collection and analysis, introduced by Strauss and Glaser in 1967 [57]. It is specifically recommended for building novel theories on complex phenomena such as, like in this thesis, the software development process in large companies. In this section we describe and discuss the rationale for using GT, its design and its implications on the validity of the results.

3.2.1 Grounded Theory design

One of the principles behind GT is the avoidance of the *hypothesis testing* approach alone, which is based on the researchers formulating an *a priori* hypothesis and then testing it with the collected data. Instead, GT challenges such an approach by proposing a bottom-up process in which the theories (and the hypotheses) emerge from the data and therefore are grounded in it. Such inductive approach is complemented with a deductive one, in which the researchers compare continuously the data in order to test the obtained theories, and they use such theories in order to drive the collection of new data (*theoretical sampling* [56]).

We have decided to start this research from an agnostic and exploratory perspective, and therefore, for such an approach, Grounded Theory was the better than formulating a priori hypotheses. There were two main reasons for starting an inductive investigation on ambidexterity rather than following existing hypotheses:

- First, as mentioned in the introduction, the existing body of knowledge in software engineering does not contain contributions on ambidexterity. Besides, in other fields (such as Information Systems and Management areas) have just started to study ambidexterity, and there is no consensus even on its definition [2]. Furthermore, the current works on ambidexterity tackle the problem on a principle level rather than taking the specific Software Engineering perspective.
- Second, and as a consequence to the above point, no theoretically sound work was found on understanding what factors were influencing the balance of short-term and long-term responsiveness in the Software Engineering research domain. The "knowledge" existing on the subject was, when the research project started, mostly anecdotic, based on opinions and beliefs that one or the other approach would work for one or the other goal, rather than being based on a thorough scientific investigation of the problem.
  This situation required what is called a *problematization* approach [58]: a critical thinking approach (one of the foundations of social science research) aimed at *demythicisation*. In other words, the process involves rejecting "common knowledge" (a myth based on anecdotal evidences or beliefs) as taken for granted, and posing the target myth as a problem to be solved. In our case, problematization was used in several cases, especially when the discussion was quite politically charged: for example, the opinion that Agile would be a good approach only for short-term responsiveness or the belief that architecture improvements are not worth the time to improve long-term responsiveness but are only considered a waste.
  The first research question reflects this problematization approach by asking the basic question "*What factors influence ambidexterity?*". In several cases, this technique have provided novel insights that are discordant from the "common knowledge" and therefore can be considered steps further in increasing the scientific body of knowledge in Software Engineering. An example among the contributions of this thesis can be found in Chapter 7, where the models of crisis point and monotonicity of Architecture Technical Debt (ATD) bring to light the hypothesis that ATD, in practice, might be unavoidable because of the co-occurrence of several factors. The problematization approach requires a first inductive investigation approach.

3.2.2 Systematic combination of inductive and deductive approach

The systematic approach given by the GT framework provided the key principles and tools to scientifically analyze and report the collected qualitative data in order to avoid a *pseudoanalysis* mentioned by Seaman, which consists on "*simply write down all the researcher's beliefs and impressions based on the time they have spent in the field collecting data*" [59]. However, although GT provides excellent support for

performing the kind of research that we wanted to do, such an approach is not perfect and it's not complete [60]: GT is not just a systematic tool that is applied and gives an objective result, but it's rather a critical thinking method that is scientific in challenging and revising theories as they emerge from the data, which helps the researchers to collect and analyze data minimizing the unavoidable bias given by human nature. Quite a lot is required from the individual researchers themselves to implement GT in the right way in order to avoid the several pitfalls mentioned in [60]. For example, the inductive approach does not imply ignoring literature, but rather motivating the inductive approach *despite* the existing literature (in this thesis case, as explained earlier, we needed a problematization approach).

GT has been developed in two main paradigms: the Glaser one [61], more focused in the inductive process and in the emerging concepts, and the Strauss and Corbin one [55], who emphasized the combination of inductive and deductive approaches, an iterative process of creating and testing theories (visible in Figure 10). The second approach is also more focused on validating theories by the systematic comparison across the data, the codes and the concepts. We followed mainly the second approach, which aims at bringing more evidence to support the hypothesis by collecting confirmatory evidence. Such an approach is also suggested in best practices of case-study research [62] and is also mentioned in two articles related to the methodologies suitable for Software Engineering research [63][59].

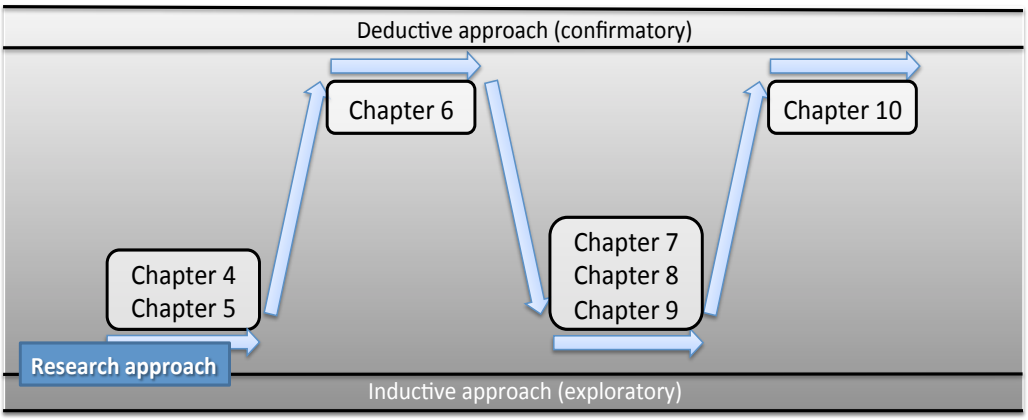


Figure 10. The systematic and iterative combination of inductive and deductive research process (blue arrows) throughout the thesis, proper of Grounded Theory.

The iterative approach can be seen throughout the whole thesis: an inductive investigation followed by the deductive and confirmatory one. For example, in Chapter 6 part of the challenges (also mentioned as inhibitors) collected inductively in Chapter 4 were tested quantitatively through a survey, in which we asked the participants to recognize such challenges in their contexts. In Chapter 7, the ATD factors and models developed during the inductive phase were tested on an in-depth case-study through the pattern matching (deductive) strategy, which is the one suggested in case-study literature [62] as the most reliable. The inductive models of increasing interest for ATD in Chapter 8 are also supported by confirming evidence emerging from other 12 cases of ATD subsequently collected across 7 companies.

3.2.3 *Validity of results and data saturation*

A special mentioning is worth on the concept, also important in the GT framework, called *data saturation*. Achieving saturation means that, with new data collected, no new codes and concepts are added to the ones already found previously. For example, saturation would be achieved with regard to Chapter 7, if conducting more interviews with software architects from new contexts would reveal that all the ATD cases mentioned in such new interviews were already covered in the previously collected

data. However, such an approach is strongly or less recommended depending on which communities we take in consideration, purists or pragmatics [60].

In this thesis, we strive to achieve as much data saturation as possible within the practical constraints given by the research problem and the research context as well. For example, we managed to achieve data saturation when investigating the crisis point model in Chapter 7: such model was investigated in all the studied contexts, and the data from the interviews showed that *all* the companies strongly recognized the pattern. However, such practice was not always feasible, for practical reasons: saturation of concepts would involve an unpredictably larger number of cases and larger number of interviews, which was not possible to obtain from the research partners (the companies involved in the project). Software engineers, architects and product owners are usually quite busy with their software development projects, and obtaining their time, both from a time and cost perspective, is quite challenging.

Considering again the two research communities writing about GT, purists and pragmatics [60], the author of this thesis aligns himself closer to the latter ones. In his view, there are two main reasons why this does not represent a drawback for the results contained in this thesis: according to Yin [62], the data collection can stop when confirmatory evidence is found in at least a new context. Runeson and Höst recommend the use of different kinds of *triangulation* [63] in order to mitigate several kinds of threats to validity: using confirming evidence from multiple sources (*source triangulation,* for example getting the same answers from architects and developers), using different methodologies (for example, employing quantitative and qualitative methods), more than one observer (when more than one person is observing an event, for example two researchers simultaneously, called *observer triangulation*) or applying different theories (*theory triangulation*). We have employed approaches that are aimed at mitigating these threats to validity, which are discussed more in depth in section 3.3.2.

The recommendations mentioned in [62], [63] weaken the heavy "completeness" constraint put by the GT framework with the complete *saturation* concept, but it still assures that the studied phenomenon is not only related to a single and exceptional case. In many cases in the various Chapters of this thesis, it is possible to see how only the quotations and phenomena that were recurrent across the studied companies were taken in consideration: a representative example is shown in Chapter 7, where a complete chain of evidence from quotations to theories is visualized using a screenshot from the qualitative analysis tool. Each code in the example was linked to more than one quotation in different contexts, which provides confirmatory evidence. This example shows how difficult (if not impossible) it would be to gather always the same answers from all the studied cases until everyone agrees on a general phenomenon. In some cases it was possible (for example, for the crisis point model, Chapter 7), but in most cases, especially when the studied phenomenon was very complex, the different contexts would bring so many different contextual factors that it would be quite difficult (if not impossible) to achieve a complete saturation. This is why, as explained with respect to [62] and [63], the studies report on the findings that were covered at least by confirmatory evidence and/or satisfying one or more triangulation criteria.

3.3 TACTICAL APPROACH: CASE-STUDIES

3.3.1 Rationale for Case-study research

The strategic choice of GT was complemented with the use of case studies as the main tactical approach in order to handle, in practice, the empirical investigation. The phenomenon of achieving ambidexterity in large software companies is strongly rooted in a very complex industrial context. Case-study research, defined as "*investigating*

contemporary phenomena in their context" [62] fits this purpose. The main design choices related to case-study aim at "*flexibility while maintaining rigor and relevance*", as explained in the following and recommended in [63].

Case study research (in this thesis involving several cases) is a widely used approach in social science research [62], but in the last years it has gained appreciation as a suitable tool for investigating software engineering topics [63], especially related to research questions concerning the study of complex systems including humans. In fact, being software engineering a highly cognitive-intense discipline, many research questions involve phenomena that are influenced by a wide number of socio-technical factors. This is even more evident when considering phenomena that present themselves in large organizations involving a large number of employees with different backgrounds, expectations and perceptions.

Other possible candidates for conducting empirical research were experiments, surveys and action research [63]. We used the canonical case study approach following [62], [63]. This choice had the following advantages with respect to the other approaches:

- *Case-studies vs experiments* - The RQs considered throughout this thesis required answers rich in details in order to provide a contribution. That is to say, they require a high degree of *realism*. This is due to the fact that the studied phenomena occur in a context (in this work, large software companies) and the boundaries between phenomenon and contexts are often not clear-cut [62]. There are two problems with the approach of isolating the phenomenon from the context in order to gain control on the factors and enabling a rigorous replication (as is done in experiments [64]): one is that contextual factors, which might be critical for understanding the phenomena, might be missed by design. Besides, it is unfeasible, with the current technologies and cost constraints, to replicate the whole development process in a laboratory.
- *Case-studies vs surveys* – Case studies differ from surveys in that surveys aim at collecting a large amount of standardized data that provide a superficial view on a subject [63]. This would not be our case in many investigated situations, since, as explained in the previous point, the RQs implied the collection on several details (high degree of *realism*) on the phenomenon in order to understand.
- *Case studies vs action research* – Action research provides quite a lot of realism [65] and it would have therefore been a good candidate in order to answer the research questions for such a point of view. However, the RQs in this thesis usually span a time frame (understanding ambidexterity between short and long-term responsiveness) that would involve the long-term involvement as action researcher, which was not possible to perform in this short-term thesis and with the research setting available for the research project. Another drawback of action research would have been to be unable to replicate case studies in other companies, which might have reduced the number of confirmatory evidence collected.

Case studies are useful for different purposes [63], especially for exploratory purposes but also for descriptive, explanatory and even for improvement: as explained on the strategic perspective (section 3.2), we have always alternated exploratory and explanatory purposes. Also, given that we have provided a solution in Chapter 10 we have used case study for improvement as well: the in-progress validation of the organizational solution uses a longitudinal case-study that shows what improvements have been reported after the implementation of the solution. For more details, Table 2 shows the various cases and their purposes: there is not a direct correspondence between the case-study and the high-level RQ of this thesis (there is, however, on a more fine-grain level, as it is shown in the chapters), since in some cases the same

study was performed for answering multiple questions, while at the same time some research questions required more than one study to be answered. For example, R1 was answered by the first case study, while R4 needed a combination of exploratory and explanatory case studies in order to be accurately understood and answered.

Table 2. The purpose of each investigation carried out in relation with the RQs

RQ	***Purpose***
RQ1 What factors influence ambidexterity?	Exploratory
RQ2 What interaction challenges affect ambidexterity?	Exploratory and improving
RQ3 What spanning activities are needed in order to mitigate the interaction challenges affecting ambidexterity?	Explanatory and Improving
RQ4 What strategic information about Architecture Technical Debt needs to be shared between architects, product owners and teams in order to manage ambidexterity?	Exploratory and Explanatory
RQ5 What organizational solution can be applied in order to facilitate spanning activities to manage ambidexterity?	Explanatory and Improving

3.3.2 Quality and Validity of the results

In this section we discuss the quality of the case studies by highlighting the various threats to validity with respect to the guidelines proposed in [63]: construct, internal, external validity and reliability.

3.3.2.1 Construct validity

Construct validity is related to the investigation device. It is important that the data collected are representative of what needs to be investigated according to the RQs. For example, by using the metaphor of *Technical Debt*, it was important, during the investigation, to make sure that we aligned our and the informants' view of what Technical Debt was referred to.

In order to mitigate construct validity threats, in most of the studies we have performed a preliminary meeting with the informants or we spent the first part of the data collection session in order to explain the terms used during the investigation and in order to align the understanding among the informants. For example, we operationalized the ATD as *architecture inconsistencies* (or alternatively *sub-optimal solutions* or in some cases *violations*, depending on the class of ATD) with respect to the current desired architecture related to a specific case.

The second approach was related to the research setting: as explained in section 3.1, thanks to the Software Center format we could assure a *prolonged involvement* [63] between the researchers and the involved participants, which assured the building of trust and therefore of an environment in which concepts from academia where spread in advance and understood by the industrial representatives, while the industrial jargon, also different in different organizations, was understood by the researchers and mapped to the same phenomenon in a more effective way.

Another approach applied in order to mitigate the construct validity threat is through the selection of the sample: we investigated the phenomenon at hand with the most knowledgeable informants for the kind of data required, and we triangulated the data from more sources [63] (different roles). For example, when eliciting ATD cases we

inquired software and system architects obtaining the best knowledge about both the desired architectures and a good explanation of the sub-optimal solutions in place.

3.3.2.2 Internal validity

Threats to internal validity are present when investigating a direct cause-effect relationship, for example if a phenomenon is influenced by a factor that has not been taken in consideration by the researcher [63]. This threat is present for many of the studies in this thesis, since we aimed at understanding factors influencing ambidexterity, factors for the accumulation and refactoring of ATD and therefore design a solution to mitigate such factors. Therefore, the cause-effect relationship has always been involved, and threats needed to be mitigated continuously.

The first approach to mitigate internal validity threats was the collection of a large amount of details for the cases studied (as explained before, this is why we strived to obtain a high degree of *realism* and therefore we needed to use case-study research), which would lead to obtain an *architecture explanation* [66] of a case. An architecture explanation is the one that does not explain causal relationship by statistically correlating two hypothetically independent events, but rather by describing how the studied phenomenon has occurred in relation to the various contextual components of the case. For example, in Chapter 7 we collected several instances of the ATD accumulation phenomenon, and by describing the several components (events) of what occurred, it was clear how the crisis point was determined by such sequence of events. In order to make this explanation more reliable, we triangulated the obtained answers from different sources (and several roles, in order to avoid the bias related to one role only), internal to the company and we always asked follow-up questions in order to probe the explanations. Also, collecting similar and confirmatory evidences across different cases, supporting the same explanation, contributed to strengthen our conclusions.

3.3.2.3 External validity

One of the major threats for case-study research is the ability to generalize the findings from the case-specific results to other cases. Generalizing to a universal theory (i.e. valid in all possible situation) is not necessarily the goal for an engineering discipline: according to [66], middle-range theories, valid to a restricted ranges of contexts, result more useful in practice. This is related to the data saturation discussed previously regarding the GT approach: it's important to find as many evidences as possible in order to promote the results to middle-range theories, but it would not be feasible to prove such theories valid universally, since the many different contexts would yield a (slightly or not) different phenomenon, which in turn would make *any* theory invalid or too abstract to give a practical benefit. The rationale in promoting middle-range theories is the need, for the software engineering communities, of having knowledge that is valid for the contexts contained in a subset of all the possible contexts: however, this should be set as large as possible, in order to increase the *range* of contexts where the results are valid. The reason for this is that the knowledge is therefore transferable (e.g. valid) to other similar contexts as well. For example, the crisis point model was recognized in 7 companies with some similarities (large, developing embedded software, etc.). This means that other companies with similar contextual factors might recognize the same phenomenon going on during their development and could therefore prevent the occurrence of the crises.

Throughout this thesis, in order to develop such middle-range theories, we have employed an analytical inductive strategy to generalize from case-studies [66]. This means that we have collected architectural explanations (see also previous paragraph) from contexts that are architecturally similar among themselves, but contain some differences: in this thesis, all of the cases are large companies developing embedded

software and having similar organizations. Only in a few occasions, we have tested the "theories" in one case that would not produce embedded software, in order to understand if the *range* of the "theories" that we were studying could be extended. The main idea is that the similarities among the cases allow the researchers to make the claims more robust through triangulation and confirming evidence, while the differences allow the extension of the findings, if similar, to new contexts.

3.3.2.4 Reliability

Reliability is important in order to understand if the data and analysis are dependent on a specific researcher or if the same study would lead to the same results by other researchers. This threat to validity is especially present when qualitative data is collected, since there are several points in which the researcher's bias might be introduce in the results.

The first approach that we took in order to mitigate reliability threats was *peer debriefing* [63], the involvement of several peers in the research activities. At a design level this was done among the different researchers involved in case, and this was also supported by the research setting given by the software center, which gave us the possibility to discuss the design early with some representative from the companies: in such way, we took in consideration the industrial perspective in order to mitigate the *halo effect* of discussing the studies considering the research problem only from an academic point of view. Peer debriefing was also performed during the analysis of data, by involving more than one author in the formulation of the coding scheme and in the activity of checking the results. During data collection, the presence of more researchers mitigated reliability threats of the study by what is defined as *observer triangulation*.

Another approach to mitigate reliability threats included the *member checking* technique [63] in which the results were reviewed by the participants in the case-study. This was a technique used in all the cases carried out in this thesis, practice that was supported by the research setting (see section 3.1), which assures, every six months, a reporting session of the results with the involved companies.

3.3.2.5 Ethical considerations

There is a number of ethical considerations that needs to be taken in consideration when performing empirical research with respect to the studied companies. The key factors, reported in [67], are:

- *Informed consent*: participants, being students or employees of a software companies, need to know that the information will be used in research. We have been meticulous with this by asking feedback on the designed or on-going investigation involving the employees before and after the case studies.
- *Benifiance to humans*: the research should not bring any harm for the participants. In case of the software companies, it was important to be sure that results shared by some of the employees would remain anonymous in order to avoid problems with managers. The same issue might be incurred by looking into artifacts and report measures.
- *Benifiance to organizations and confidentiality:* by publishing results, competitors of the investigated software companies could take advantage of the published data. In order to avoid this issue, we have always anonymized the companies with fictitious names and asked for their consent to publish the paper in the submitted form.

3.3.3 Case selection

In case study research, it is particularly important to select cases that have suitable characteristics for answering the RQs. In particular, the selection strategy needs to have a rationale on the number of investigated cases and on the replication strategy. In all the studies included in this thesis we usually conducted multiple case studies (with the companies described in 3.1.2), which means that we involved several companies in the study. The rationale was that by doing so we would increase the reliability and validity of the results (see sections 3.2.3 and 3.3.2).

We usually followed a replication of (hypothetically) typical cases, in which we applied variation on context elements that would possibly bring new insights according to the *maximum variation* principle [62]. In some cases, for example when including company E (not an embedded software company), we aimed at understanding if such case was deviant (very different from the embedded domain), or if the results found for the other companies would apply to company E as well. Sometimes, in order to increase triangulation, we used *embedded cases*, in which we involved sub-organizations of the same company as a case itself (for example for company C). We made sure that such sub-cases had the same context as the other sub-cases. This replication strategy is visible in Table 3.

In some cases we also performed case studies that lasted longer than one phase only. Each phase can be seen as a different case. This was done in order to follow the GT iterative strategy: usually wanted to evaluate our exploratory hypotheses, obtained from the inductive phase, with an explanatory (or evaluative) phase in which we were collecting further confirmatory evidence.

The distribution of the companies (described in section 3.1.2) with respect to the following Chapters of this thesis is shown in Table 3. Such table highlights also how many phases each Chapter involved. In some cases, we represent a company between brackets if the results from such company have not been published yet (and therefore are not present in this thesis, but are part of the data collection).

Table 3 Number of phases and companies involved in the data collection with respect to the chapters

Chapter	***Number of phases***	***Companies involved***
4	1	A, C, E
5	1	A, C, E
6	3	A, C, E
7	4	A, B, C, D, F
8	4	A, B, C, D, F
9	1	A, B, C, D
10	2	A,C (D,E,F)

3.4 DATA COLLECTION METHODS

In this thesis we conducted a large number of interviews. The numbers are reported in Table 4. We also compensate the qualitative data collection with quantitative data collection, as visible in the table.

Table 4 Number of data collection sessions, type, number of informants and hours of qualitative data collected and analyzed with respect to the research process. "Additional" are the interaction with informants that were not recorded but occurred in spontaneous ways. Each "phase" (also called "research sprint") lasted approximately 6 months.

Phase	Number of data collection sessions	Qual./ Quant.	Number of informants	Hours recorded and analyzed
1	7	Qual.	7	14
2	38 survey answers	Both	38	-
3	9	Qual.	14	15
4	3 + 15 survey answers	Both	40	12
5	7	Qual.	25	11
6	3 + 12 survey answers	Both	19	10
7	10	Both	27	21
8	9	Both	25	19
Additional	(informal interviews)	Qual.	N/A	N/A
Total	48 Qual. + 65 Quant.	-	72 Qual.* + 65 Quant.	102

* Some of the informants participated multiple times in the interviews.

3.4.1 Interviews

Interviews have been one of the main investigation device used in this thesis. We mostly used semi-structured interviews, in which we had pre-defined topics to be covered and a few opening questions. Rather than having specific questions for the whole session, we outlined a list of goals that needed to be covered in the interview and we let the discussion flow. The questions were mainly related to following-up questions on interesting topics brought up by the interviewees or attempt to probe the interviewees' statements with concrete facts. Semi-structured interviews are very useful for exploratory purposes [68], while for more confirmatory and evaluation purposes structured interviews (or questionnaires, described later) are more suitable. The interactive nature of the interview was useful to catch and develop interesting and unexpected themes brought up by the interviewee, which could not have been possible with other methodologies such as a questionnaire.

Informants were chosen on the basis of their role and expertise and their relevance with respect to the investigation performed. All informants were senior engineers or managers. The interview protocols had usually fixed questions about the informant's role and context (when not known in advance), and open-ended questions to bring out experiences and opinions. During the interviews we tried to follow the natural flow of the discussion, with questions injected to cover all themes. Since the process of interview included also a partial analysis, the successive interviews contained references to ideas captured in previous sessions.

Interviews were useful because the participants could describe processes and organizational features (useful for understanding the context), or the interviewee's

perspective on the studied phenomenon. Since there are many factors influencing ambidexterity, interviews contributed to provide a broad landscape of factors.

Interviews were also useful to understand and map the terminology in place at the studied site and perhaps to disseminate the academic vocabulary (or, on the other hand, to update it), especially at the beginning of the project. This was especially important in order to mitigate the construct validity threat [63] (see section 3.3.2.1): we always wanted to make as sure as possible that the investigation device used in the data collection was understood by the interviewees. For example, in the Technical Debt investigation it was important to probe the understanding of the employees on the used terminology, e.g. what was considered interest and principal of the debt.

Interviews provide additional tacit information: gestures, voice tone, facial expressions and nonverbal communication in general play a key role especially in the interpretation of the mere words [68]: this facilitates and makes the analysis more reliable, for example it allows the researcher to weight the answers or might suggest the right methodology to ask further questions. Although such additional information cannot be recorded and systematically tracked in the chain of evidence, understanding the employee is part of the interpretive role of the researcher as much as the data analysis itself. Such interpretation includes a bias threat introduced by the researchers in the interpretation of the data, but not applying it and weighting the evidences all at the same level of reliability would have suffered from the opposite bias threat problem. Therefore, we decided to apply the interpretation step in a cautious way: based on the tacit information, we filter or weight data according to if they appear reliable or not.

Another useful property of interviews was the ability, for the researchers, to probe the statements of the informants. In some cases, such as in a questionnaire, a statement needs to be interpreted and follow-up questions are not always possible. However, in an interview, it's possible to ask for more evidence to support statements, which was very useful when evaluating factors and models developed during the exploratory interviews (hypotheses).

3.4.2 Questionnaires

In three cases we have used the questionnaire as an investigation device. The first is visible in Chapter 6, the second one was used in Chapter 9 and another one was used in Chapter 10. All the questionnaires were used for confirmatory purposes, in order to evaluate previously obtained findings. This a suitable way to use questionnaires, and it gives the results more reliability, since it provides more confirmatory evidence on a specific phenomenon, but also provides more triangulation of the answers among different respondents [63].

The preparation of the questionnaires have always followed a careful design, in which we followed the best principles of survey design [69]. One of the major challenges in this kind of approach is the threat to construct validity (discussed in section 3.3.2.1). In the first one, we conducted two pilot runs, one with academics and one with multiple participants from industry, in order to be sure that the device was constructed in the right way and the concepts were understandable and represented what we wanted to investigate. In the second and third surveys, we spent one or more hours before the survey with the participants in order to make sure that they all aligned on the meaning and purpose of the questions.

3.5 DATA ANALYSIS METHODS

Most of the research conducted in this thesis is based on qualitative investigation: the reasons is that, as explained in previous Chapters, the main focus of this thesis is to understand how several teams and individuals, with different competences and tasks,

would need and consume information in order to coordinate for balancing the development of short-term vs long-term software solutions. Such activity is complex because it involves many human interactions and depends on a number of factors.

Qualitative research was chosen because *"Qualitative research methods were designed […] to study the complexities of human behavior (e.g., motivation, communication, understanding)"* [59]. Given the large amount of humans involved in the studied phenomena and the complexity of the context, this was the best choice. Therefore, even if we have applied, in some cases, quantitative methods (explained in the specific Chapters), we will put the focus on explaining how qualitative research was performed.

We analyzed the data using a combination of qualitative and quantitative approaches, which contributed to apply methodological triangulation, as recommended in [63]. Qualitative analysis gives in-depth understanding of a process or a system, while quantitative analysis is more useful for strengthening results, to test hypothesis and to find information related to respondents or contexts with specific properties.

3.5.1 Qualitative Methods

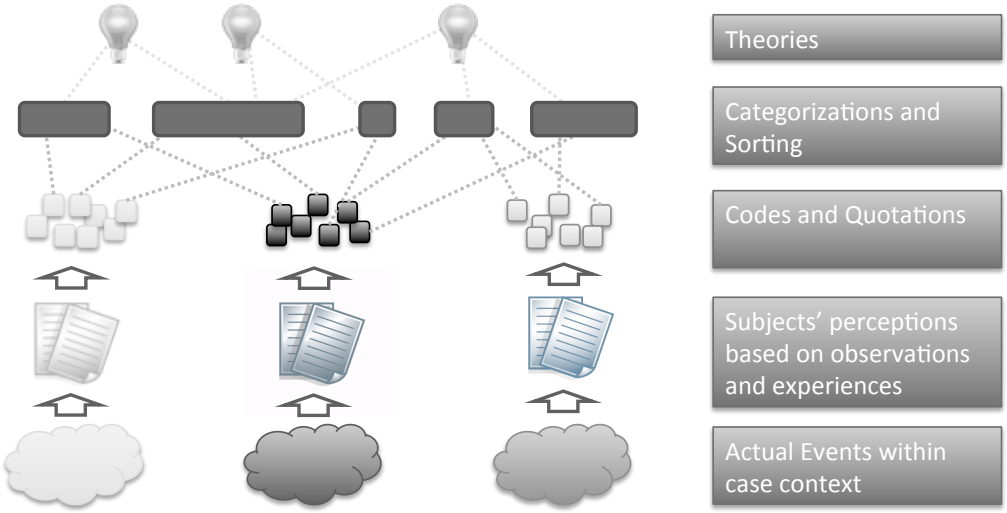


Figure 11. Qualitative data collection and analysis process

The qualitative analysis process is described in Figure 11: from the actual events we could access the practitioners' experiences, in the form of written transcriptions. We coded the text using a tool for qualitative analysis, which allowed us to maintain traceability between the data and the produced codes. A screenshot of a concrete use of the tool is showed in Figure 12, in Chapter 7. The codes were categorized and sorted in order to link, compare and combine them into an organic representation of the investigated data, which led to identify novel theories or to confirm existing knowledge. We conducted the following steps, also showed in Figure 12.

Open Coding

As a first step for inductive analysis, we have used open coding: this process is necessary to create "concepts [that] are building blocks of theory" [68]. We analyzed the audio from interviews and the text from the open-ended answers in the questionnaire, slicing it into small pieces and linking the relevant ones to quotations and substantive codes. This was part of the open coding, in which we followed Strauss and Corbin's method [61]. Some concepts were the result of in-vivo coding (quotations directly converted in codes), others were syntheses, made by the researcher, of the overall concept or phenomenon expressed by the informant. The link between the quotations and the codes was always traced thanks to the use of a Qualitative Data

Analysis (QDA) tool, as shown in Figure 12. Then, the codes were grouped into categories.

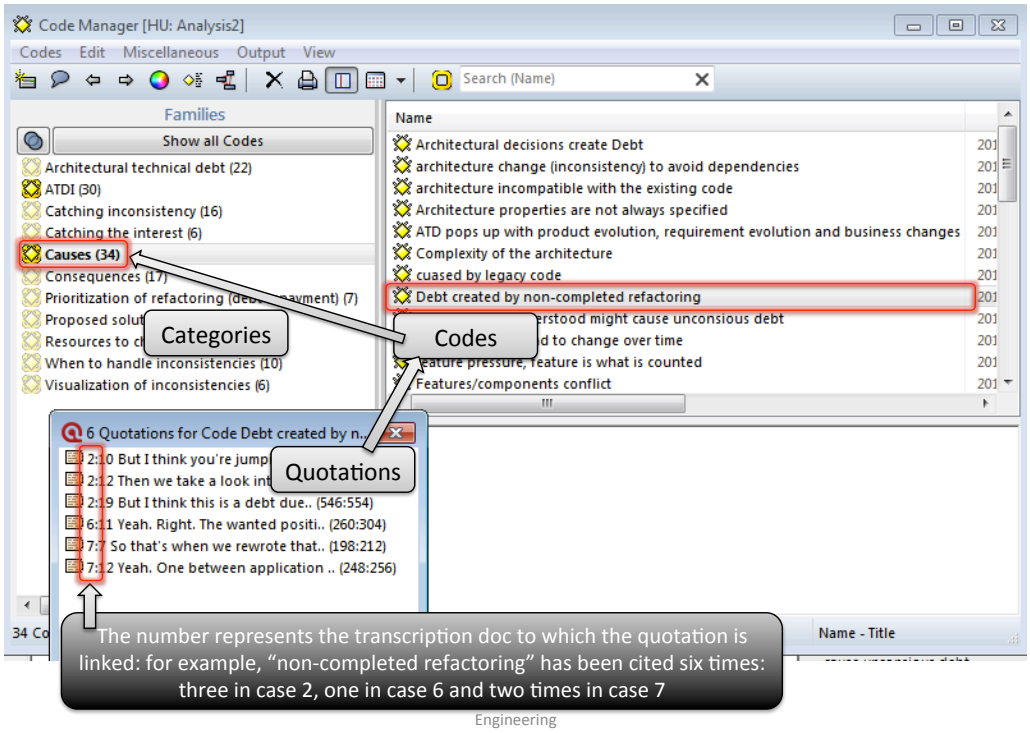


Figure 12. Qualitative analysis process using a QDA tool in order to keep track of the chain of evidence: from multiple quotation, collected and transcribed from the informants, to codes and categories.

Axial Coding

The codes and categories were compared through axial coding in order to highlight connections orthogonal to the previously developed categories. Axial coding is a useful way to analyze the data in order to find relationships and patterns related to a specific dimension (axis) of the data. This technique has been used in various studies in this thesis. A good example is reported in Chapter 7, where the *time* axis was used in order to relate the various factors among each other, in order to understand their sequence over time and therefore to relate their influence on the a studied phenomenon (in such Chapter it was the accumulation of ATD over time).

Deductive analysis

Deductive qualitative analysis was performed when a structure was already in place and we wanted to fit the data into such structure rather than creating new categories (however, new codes were always created). Such analysis is useful for evaluation purposes: first of all, such analysis shows what confirmatory evidences have been collected with respect to a give category or concept. As an example, we might take a factor in Figure 12: in the QDA tool we can see how many quotations and from which case the quotations are related to the concept. This is important for triangulation purposes [63] and in order to assess data saturation [61]: the more evidences (quotations) were found related to the same factor, the more confirmed such factors was considered by the researchers.

Other than the confirmation at a code level, another special case of deductive analysis was done, which is called *pattern matching*: such analytical strategy is based in having an hypothetical pattern coming from the exploratory data (for example, a model of a sequence of events) and then having another, new case that is analyzed separately. The last step of this technique consists of superimposing the two models

and to compare if the second model is matching the previous one. In such case, we say that the *pattern* is *matched* and therefore we have confirmatory evidence of the entire phenomenon and not only on the single factors. Such strategy is one of the most recommended in qualitative analysis and is considered one of the most strong strategy for confirming exploratory phenomena [62]. It is also a very expensive strategy, since the researchers need to collect new cases and study the overall phenomenon in order to see if the newly found pattern matches the exploratory one. We have especially employed this technique in Chapters 7 and 8. In Chapter 7 we specifically show how the found patterns were matching.

Document analysis

In order to corroborate the findings with source triangulation, we conducted, when possible, an analysis of the documents related to the studied phenomenon. This was the case when studying Architectural Technical Debt, in which we especially focused on understanding the mismatch of the code from the architecture (in which case we would be able to find and understand the *sub-optimal solutions*).

Another set of documents studied were the architectural improvements databases that (some of) the companies were keeping. The entries in such database were considered ATD items (recognized sub-optimal solutions of the code with respect to the architecture), or future architecture features that were planned for future development. Examining these documents was considered especially useful for the collection of confirmatory evidence when we were compiling the taxonomies in Chapter 8.

3.5.2 Quantitative Methods

In the three surveys, we have usually employed simple descriptive statistics in order to visualize a phenomenon (for example, comparison of answer frequencies), and in the first one we could employ the chi-square test. Such test was usable for nominal data, but in the second case (Chapter 6) such method could not be employed because the sample size was too small and it would not give reliable results. In the first questionnaire, we also applied the more advanced ANOVA approach during the analysis: however, such approach was not considered very reliable because of the limited sample size to which it was applied (as reported in [70], not included in this thesis since an extended version was accepted for publication in a journal, where the results from ANOVA were not reported).

4 FACTORS INFLUENCING AMBIDEXTERITY

In this Chapter we investigate the factors influencing Ambidexterity. The speed offered by agile practices is needed to hit the market fast, while reuse is needed for long-term responsiveness. We presents an empirical investigation of factors influencing speed and reuse in three large product developing organizations seeking to implement Agile practices. We identified, through a multiple case study with 3 organizations, 114 business-, process-, organizational-, architecture-, knowledge- and communication factors with positive or negative influences on reuse, speed or both.

The contributions in this Chapters are a categorized inventory of influencing factors, a display for organizing factors for the purpose of process improvement work, and a list of key improvement areas to address when implementing reuse in organizations striving to become more Agile. Categories identified include good factors with positive influences on reuse or speed, harmful factors with negative influences, and complex factors involving inverse or ambiguous relationships. Key improvement areas in the studied organizations are intra-organizational communication practices, reuse awareness and practices, architectural integration and variability management. Results are intended to support process improvement work in the direction of Agile product development. Feedback on results from the studied organizations has been that the inventory captures current situations, and is useful for software process improvement.

This chapter has been published as:

Martini, A., Pareto, L., Bosch, J. *"Enablers and Inhibitors for Speed with Reuse"*, published in proceedings of Software Product Lines Conference, SPLC 2012 [71].

4.1 INTRODUCTION

There is an ongoing paradigm shift in industry from plan driven to agile product development, with increased speed, -throughput, and -customer value as common business goals. The shift presents challenges when combined with other paradigms, such as Software Product Line Engineering. Both industry and academia have recognized the importance of this combinations (APLE), and academia has been successful in theoretically prove their principles to be complementary [72]. However, the combination presents challenges in all areas of product development (business models, organizations, processes, and product architecture), but especially in the Domain Engineering related practices [72]. Here the ideals of agile development (e.g. big-design-upfront avoidance, and people over extensive documentation) seem, at a first sight, to collide with best practices of software reuse (e.g., domain engineering, variability management, components). Academia is currently exploring feasible combinations of reuse- and agile practices [72]. Companies explore hybrid forms. Reuse researchers evolve best practices for reuse to new situations [72]. Agile researchers are to an increasing degree recognizing the importance of maintaining "architecture runway" in their software processes [73]. However, too little empirical evidence has been provided about the effective combination of reuse and agile practices [72]. Moreover, the focus is primarily put on introduce Agile into well-established SPL rather than on the implementation of software reuse into an agile-oriented company [72]. Besides, many emerging combinations of agile and reuse practices are primarily exploratory hybrids seemingly unguided by theory on how respective practices interfere or support each other [72]–[75]. The premise of this paper is that software process improvement activities in the direction of agile practices need to be informed by an understanding of these

interferences (see e.g. Leffingwell [76]), or good intentions may worsen rather than improve fulfillment of business goals.

The purpose of the research presented in this paper is an analytical framework that relates agile and reuse practices to the measurable phenomena *reuse* and *speed* in the context of large-scale product development. By reuse, we mean the reuse of software assets to decrease product development and maintenance costs. Speed stands for an organizational unit's ability to react quickly to requests with value for another partner. By large-scale product development, we mean product development in organization with several hundreds of developers, long-lived product families (5 years or more), and evolving product lines.

The paper presents an exploratory qualitative multiple-case study, in which experiences from reuse in three large product developing organizations (A, B, C), were analyzed to recognize technical and organizational factors with influence on reuse or speed, and their relationships.

The precise research questions addressed are the following. For company A, B, and C:

Q_1: Which factors influence reuse?

Q_2: Which factors influence speed?

Q_3 Which factors influence both?

Q_4: Are influences positive, negative, or both?

Q_5: How can we decide which factors to address when implementing agile product development?

The paper's main results are an inventory of influencing such factors, displays for categorization and presentation of large numbers of such factors, and recommendations for the use of such displays to guide software process improvement work. The intended use of such inventories and displays is to help organizations to optimize the return on investment (ROI) of research and development (R&D) activities, through directing improvement activities to areas where they have the most positive impact.

The paper is organized as follows. We present a conceptual framework, our research design, the resulting displays and their principal usage. We then exemplify the use of the displays by drawing inferences from our results and the displays from our three cases. The chapter ends with discussions on the generality, validity and limitations of our results, discussions of related works and our conclusions.

4.2 CONCEPTUAL FRAMEWORK

A conceptual framework [77] for our analysis, defining categories and relationships of interest to our research questions, are given in Figure 13. We seek to identify factors that influence reuse or speed (influencing factors), and ultimately also a company's return on investment on research and development activities (ROI of R&D). We are also interested in the polarities of these influences, i.e., whether influences are positive or negative (+/-). We recognize that reuse, speed and ROI are measurable phenomena (Qualities), whereas factors in general are not. We also recognize that reuse, in itself, has high influence on speed.

The framework emerged through analysis of interviews, bottom-up categorization of factors found and by superimposition of van der Linden's Family Evaluation Framework [47]. It is used as backbone for presentation of analysis results, thus we explain its constituents in detail.

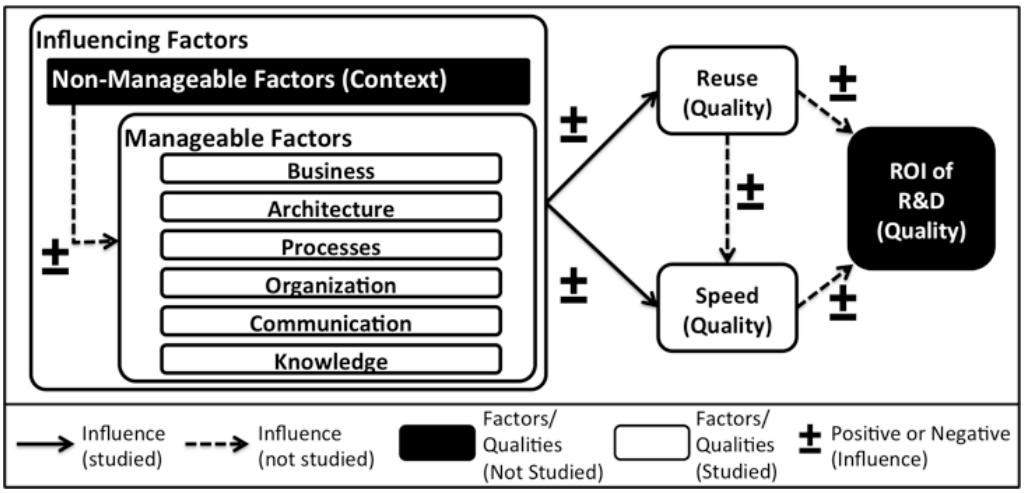


Figure 13. Conceptual Framework

4.2.1 Influencing factors

Influencing factors can be divided into *manageable factors* that a company can control, and *non-manageable* factors belonging to the company's context. Manageable factors are divided into *business-*, *architecture-*, *process-*, *organization-*, *communication-*, and *knowledge related factors*. Non-manageable factors include legal, political, technological, social, and market factors. This paper is restricted to the study of manageable factors, as these are more important than non-manageable factors, from an improvement perspective.

Our framework recognizes that influencing factors sometimes hinder reuse (R-), hinder speed (S-), boosts reuse (R+), boost speed (S+), and sometimes have combined such influences (⟨R-, S-⟩, etc.). For instance, systematic platformization of features in a new product increases reuse, but reduces development speed.

4.2.2 Reuse

The term reuse is, in its everyday use, heavily overloaded, thus when interpreting interviews, reasoning about influences, and proposing improvement actions, precisions is motivated. To this end, we use the following definitions.

Reuse is the degree to which assets occur in several products. We characterize reuse along three dimensions: the *asset dimension,* the *granularity dimension* and the *purity dimension*. In the asset dimension, we distinguish kinds of reuse with respect to the level of concretization of the software: *code reuse*, *model reuse*, *specification reuse,* and *design-structure reuse* in increasing order of abstraction. In the granularity dimension we distinguish, *module reuse*, *component reuse*, *platform reuse*, and *subsystem reuse* in increasing order of size. In the purity dimension we distinguish *ad-hoc* (clone-and-own) reuse, and *managed reuse* (with respect to ownership, transparency, replication, and other qualities).

Notice that an influencing factor may bear differently on these kinds of reuse. For instance, introduction of domain engineering will increase managed platform reuse, but decrease ad-hoc reuse.

4.2.3 Speed

Similarly, our framework recognizes different kinds of speed, which are presented in Figure 14. There are 3 *kinds of speed*: *first deployment speed*, *replication speed*, and *evolution speed*. These in turn have *constituent speeds*: *decision speed* and *development speed*. Each kind of speed is defined as the duration between two key

events, which are different for each kind: the leftmost column in Figure 14 shows which. The *first deployment speed* involves the time for deciding whether to develop a product with some new functionality (Product 1), and the time spent on actually developing the product. Similarly, *replication speed* involves the time for the decision to reuse functionality in some product (Product 2), and for actually developing this product with reuse of code and artifacts from product 1. *Evolution speed* involves the time used to decide whether an incoming evolution request should be addressed or not, and the time for actually developing it.

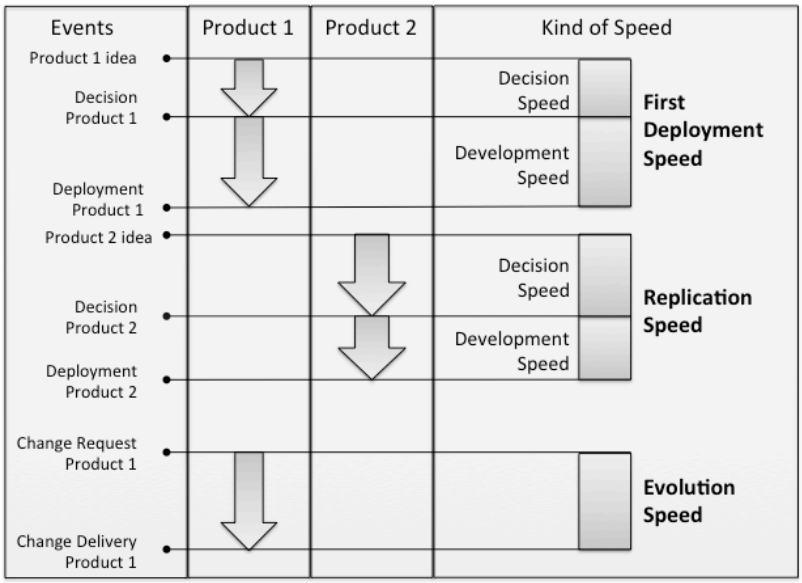


Figure 14. Different kinds of speed

Notice that the different kinds of speed are important in different lifecycles phases in product development. The *first deployment speed* is important in the very beginning of a new function, while the *replication speed* is important later when a successful function diffuses to other products. *Evolution speed* is important when the product is new on the market (to quickly fix teething troubles experienced by early adopters), while the product is expanding on the market (to compete on quality and new features), and when the product is coming to age (to allow upgrades of the product with modern components and features).

4.2.4 ROI of R&D

In the light of our framework, it is clear that different practices for reuse and speed influence a company's ROI of R&D in intricate ways, leading to difficult tradeoffs between expected long-time and short-time payoffs. Although the study of these relationships lies beyond the scope of our research questions, they motivate our research questions and are, for this purpose, recognized.

4.3 RESEARCH DESIGN

We conducted a multiple-case study in three Swedish companies involved in large-scale product development. Semi structured interviews were conducted for data collection, whereas data analysis followed the Grounded Theory approach [55]. The next sections describe the research setting and the research process.

4.3.1 Case descriptions

Our cases were software development teams within three large product developing companies, all with extensive in-house embedded software development. All of them

were situated in the same geographical area (Sweden), but they were active on different international markets. Their characteristics are summarized in Table 5.

Our first case was a team inside a manufacturer of utility vehicles. The unit produced middleware for services such as navigational support. The unit served different projects, all of which provided requirements to the unit; the resulting middleware was shared among all the projects. The unit was working in parallel with another team in a partially agile environment. The unit's product (the middleware) was developed using modeling tools and code generation. Their processes used iterations of about 10 weeks, after which the middleware was delivered to the testers. Testers were located in a separate team that verified and validated the output. The unit outsourced some components and had recently been working on implementing Software Product Lines, which however, in the end (due to market pressure and internal decisions) resulted in independent products.

Our second case was a team within another manufacture of utility vehicles; the team developed a communication subsystem for one of their product lines.

The unit was responsible for several large subsystems, developed in independent projects. Implementation of agile processes had been initiated, while team composition still followed formal and standard project processes. Models were not used, but rather extensive formal documentation. Outsourcing was absent, whereas some Open Source components were used. They employed, in some cases, *design-reuse* for small components, and they utilized a light framework of libraries extracted from previous projects and extended by developers (*ad-hoc reuse*).

Our third case was a team within a company developing field equipment. The unit developed a component for equipment control used in three product lines. The unit had 20 developers, organized as two cross-functional teams employing agile practices. The requirements on the component and its design were received from an external systemization team. The component was developed using traditional coding without support from modeling or variability tools. The unit tested the component internally, after which it was integrated in the three products to be verified and validated once again against the respective product requirements. The unit represented a pilot project in software reuse where common functionality from three products was commoditized. This had meant that two of the three projects reusing the component had to replace previously owned and non-shared components with a shared one.

Table 5. Context description of the cases

	Case A	**Case B**	**Case C**
Case (Unit of analysis)	Development team participating in several projects	Development team participating in several projects	Development team serving other units
Product description	Middleware for on-board information systems.	Communication system, signal elaborating system, GUI system and customized OS platform	Common component for equipment control shared by 3 products.
Product lifecycle	6-7 years	> 15 years	~15 years
Business Model	International Markets	International Markets	International Markets
Processes	• SCRUM based implementation – Several weeks iterations • Input: requirements from different product owners • Output: deliver a *platform* to testers to be validated and verified	• SCRUM based implementation – Several weeks iterations • Input: specification System Engineers • Output: delivers a *subsystem* to separate unit of testers to be validated and verified	• SCRUM implementation – few weeks iterations • Input: white box specification from System Engineers • Output: deliver a component to be integrated by other units developing other products
Organization	• 20 developers (including architects) • Cross-functional teams • Testers in a separated team • Requirement Engineers in separated business unit	• 2-25 developers (depending on the project, including architects) • Organized in functional teams • Testers in separated team • System Engineers in separate team	• 20 developers (including architects) • Cross-functional teams • System Engineers in separated team
Outsourcing	Some components outsourced	No outsourcing	No outsourcing

4.4 RESEARCH PROCESS

Figure 15 outlines our research process. We followed a grounded theory approach, with inductive analysis of qualitative data extracted from the interview [68]. The steps of the research process are described below.

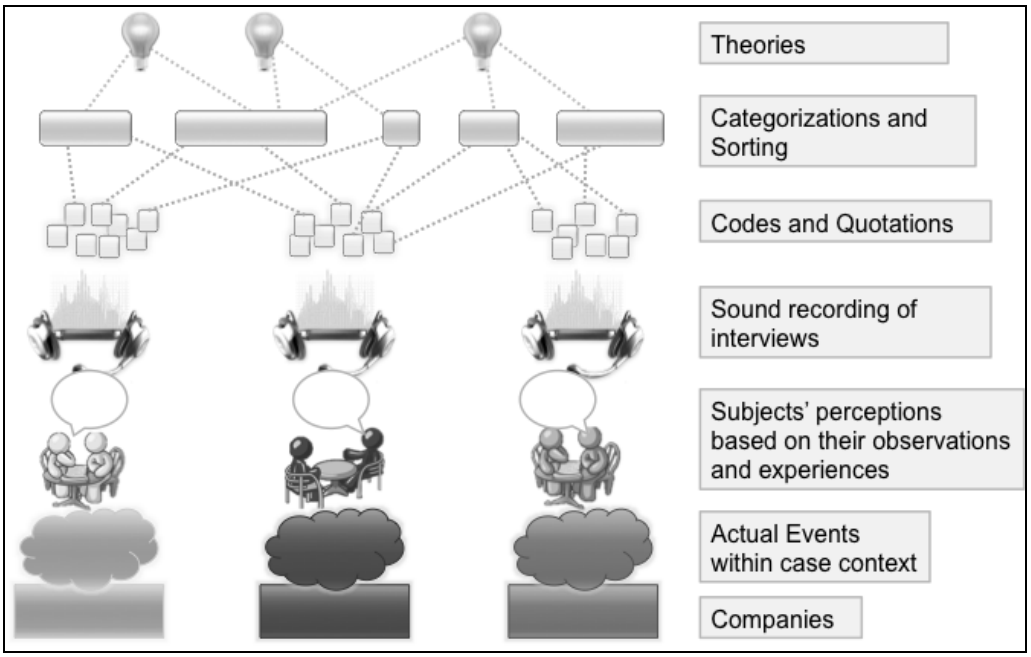


Figure 15 Our research process

4.4.1 Interviews

We conducted 7 semi-structured 2h interviews with 1-4 informants from each company.

Informants were chosen on the basis of their role and expertise. All informants were senior engineers or managers, and had experience with the implementation of agile processes. For case A we interviewed an architect and a line manager from within the same project. For case B, we interviewed four developers from four different projects, some of whom had also architect or product line management roles. In case C, we interviewed one informant with the role of both senior developer and architect.

The interview protocol had fixed questions about the informant's role and context, and open ended questions to bring out experiences and opinions related to reuse and speed: details are available in a technical report [78]. Interviews followed the natural flow of the discussion, with questions injected to cover all themes.

4.4.2 Data analysis

Out of the seven interviews, four were selected for detailed analysis and coding. This was to obtain balance between the three companies, and because some interviews turned out to be more relevant to the research questions than others.

We first analyzed the audio, slicing it into small pieces and linking the relevant ones to quotations and substantive codes. (This task was carried out by the first researcher and checked by the second one) This was part of the *open coding,* in which we followed Strauss and Corbin's method [55]. In this phase, the first researcher summarized and interpreted the informants' words, which included different methods. Some factors were the result of in-vivo coding, where we quoted the interviewee; this was done in cases where the concept was well explained, e.g., F-111: *"Improvements depend on leaders mindset, open to listen and recognize strengths and weaknesses"*. For other factors we summarized from the source, e.g., F-37, "Long warming up

periods for consultancy" summarized *"[…] is very hard for a consultant coming in […] we see that we have a half-year initial period just for [tool name] tool; it's another half year or year for the product to come into that to know it on a pretty good level […]"*. Yet other factors, are referring to many statements, which are showing the specified clear issue, only if considered together, e.g., F-3: "Business side afraid of upfront investment for an SPL". Where applicable, we used the terminology defined in section 2. We also applied *selective coding*: we discerned, from the 369 total codes, only the ones that we believed to be enablers or inhibitors of reuse or speed.

We then categorized codes according to our research questions, i.e., with respect to positive or negative influence on reuse or speed. This categorization relied on the informants' and researchers' experiences, relationships reported in literature, and logical implications. The categorization revealed the need for several hybrid categories for special influences, such as influencing several kinds of speed with different polarities: for instance, "Lack of detailed documentation" has a negative influence on *1st deployment speed* but a positive influence on the *Replication Speed*.

These 9 categories were then arranged into a visual display that we refer to as a *factor map*. To allow feedback from each organization, and to allow reasoning about differences in contextual factors for the three cases, distinct factor maps were created for case A, B, and C; these were also coalesced into a common factor map (given in Table 7 below). We also categorized codes with respect to improvement areas. Categorization was done inductively, with attention to the BAPO framework [47], and eventually led to the taxonomy of improvement areas.

4.5 RESULTS

This section presents the *influence categories* found, the *factor maps* and how these can interpreted and used.

4.5.1 Classes of influence

Figure 16 shows the 9 influences categories, and how these relate to our conceptual framework. Influence may be positive, or negative, which leads to six straightforward categories: *reuse enablers (R+)*, *reuse inhibitors (R-)*, *speed enablers (S+)*, and *speed inhibitors (S-)*, *bi-inhibitors* (BI-), and *bi-enablers* (BI+).

Other factors are inhibitors and enablers at the same time: *inverse factors (IF)* (in the sense of inverse relationships), that inhibit speed and enable reuse or vice versa; *ambiguous factors (AF)* that both inhibit and enable reuse, and those that both inhibit and enable speed.

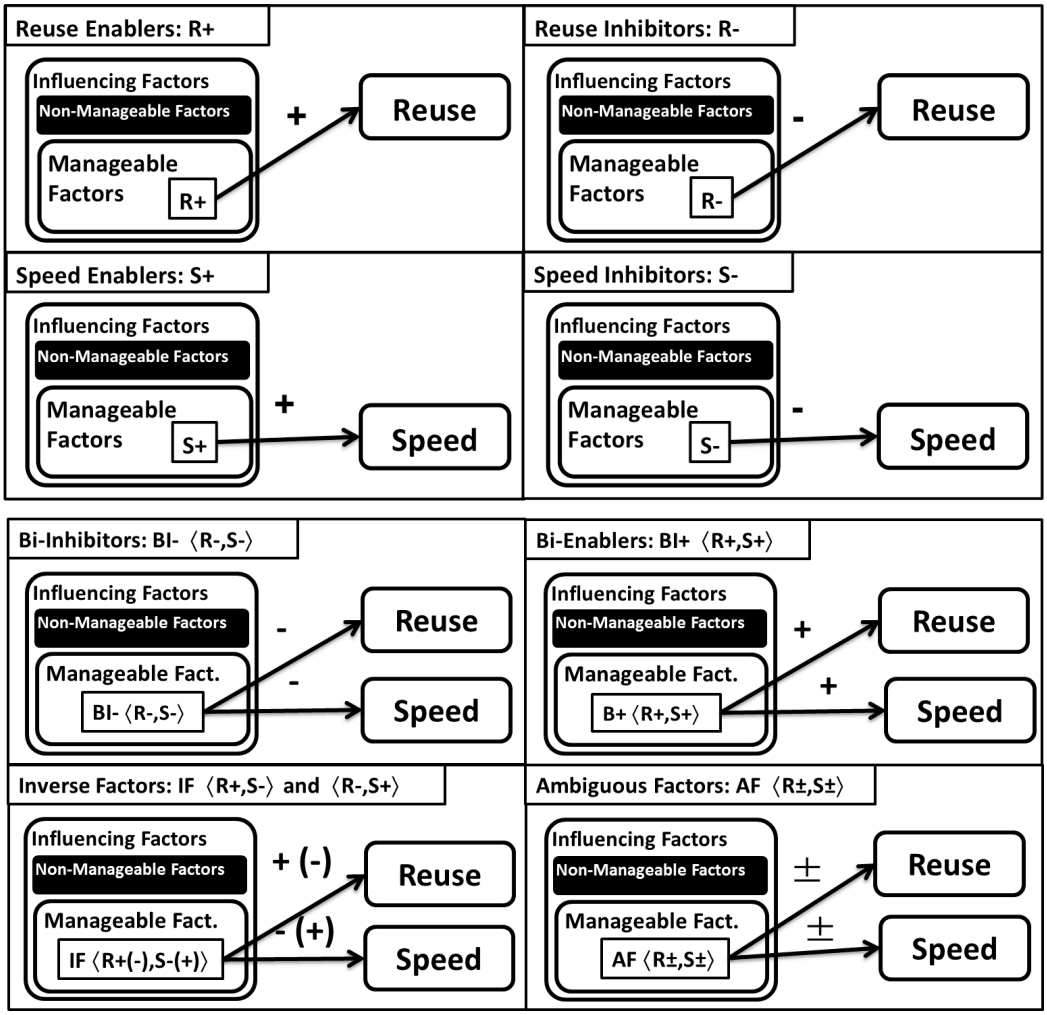


Figure 16.Influence classes used in the factor maps

4.5.2 Factor maps

A *factor map* is 3x3 grid of influence categories, populated with concrete factors. The inhibitors are placed to the left (*reuse inhibitors, bi-inhibitors*, *speed inhibitors)*, enablers to the right (*speed enabler*s, *bi-enablers*, and *reuse enablers*), and *inverse* and *ambiguous factors* in the middle.

Table 7 is an example of a factor map resulting from our case. It shows the layout of influence categories, and gives representative samples of each, from our study. Notice that Table 7 only includes some of the altogether 144 factors found in our study. The full set of factors is available in a technical report [78].

The factor map structure (Table 6) is designed to support quick recognition of *good*, *harmful* and *complex* factors. Good factors (*R+*, ⟨R+,S+⟩, S+) influence only positively the development process in terms of reuse and/or speed, and are found in the right column. Harmful factors, are only causing difficulties, and are all found to the left. Complex factors, in middle influence reuse and speed in intricate ways.

Table 6. Factor map layout

HARMFUL FACTORS	COMPLEX FACTORS	GOOD FACTORS
Reuse Inhibitors	**Inverse Factors**	**Speed Enablers**
Bi-Inhibitors	**Ambiguous Factors**	**Bi-Enablers**
Speed Inhibitors	**Inverse Factors**	**Reuse Enablers**

Table 7. Example of a factor map

Reuse Inhibitors (R-)	***Inverse Factors (IF2) ⟨R-,S+⟩***	***Speed Enablers (S+)***
F-9. Different products initially managed with a branch, then it became too late to merge the branches into a SPL (*RPK-K*), case A F-20. Reuse not supported from the PL - individual initiative (*BRA-B*), case B F-35. One of the products had a longstanding, dated process of verification and validation which suffered when applied to the new component (*VV-P*), case C Full set of *R-* factors: F-1 to F-36 [8].	F-100. Customer functionalities are preferred by developers over architecture (*PR-P*), case A F-103. Risk in planning reusable components - lost time for the first client and having the wrong variation points in the future (*VM-A*), case B Full set of ⟨R-,S+⟩ factors: F-99 to F-103 [8].	F-80. Co-located teams (*AG-P*), case A F-81. Weekly prioritization of the functionalities with the customer (*AG-P*), case A F-83. Changing programming language improved productivity (*TEC-K*), case B
Bi-Inhibitors (BI-) ⟨R-,S-⟩	***Ambiguous Factors* (AF) *⟨R±,S±⟩***	***Bi-Enablers (BI+) ⟨R+,S+⟩***
F-58. Satellite unit auto prioritizes, have different masters (*OUT-O*) case A F-59. Consultancy has knowledge of others' ways of working but don't have specific knowledge of the product. (*CWS-K*), case A F-70. "White box" specification from systemization (*ICP-C*), case C Full set of R- S- factors: F-55 to F-70 [8].	F-112. The same employees implement functionalities and architectural improvements (*AG-P*), case A F-110. Lack of detailed documentation (*DOC-C*), case A F-111. Improvements depend on leaders mindset (open to listen and recognize strengths and weaknesses) (*ICP-C*), case A Full set of R± S± factors: F-104 to F-114 [8].	F-89. Incentive from management side to developers' proactivity: communication of small improvement through meetings (*ICP-C*) case A F-92. Reuse of the design for small components (*RPK-K*), case B F-93. Developers have the will to take part in the systemization (*ICP-C*), case C Full set of R+S+ factors: F-88 to F-94 [8].
Speed Inhibitors (S-)	***Inverse Factors (IF1) ⟨R+,S-⟩***	***Reuse Enablers (R+)***
F-41. Standard project structure doesn't fit some kinds of projects (*PS-O*), case B F-53. Layer architecture against functional domain of the component caused delays in verification and validation of the component against all the layers (*AI-A*), case C Full set of S- factors: F-37 to F-54 [8].	F-96. Functionality requests from different projects (*PCF-C*), case A F-97. The company has a conservative mindset and tends to keep old assets – this often needs extra code to handle them (*PCF-C*), case B F-98. Massive use of documentation (*DOC-C*), case B	F-71. Reuse of models (*RPK-K*), case A F-74. Frequent meetings and agreement between Developers and System Engineers discussing the benefits of Reuse. (*ICP-C, ARB-K*), case B F-75. Long term plan: having more common components integrated in the different products (*ARB-K*), case C Full set of R+ factors: F-71 to F-79 [8].

Based on this categorization, we can relate the characteristics of an organization to the factors on the map. If there is a correspondence with a factor belonging to the *good factors*, these factors should be maintained or further improved. On the contrary, if a property can be linked to a *harmful factor*, the company should consider handling it. As for the *complex factors,* they may be taken into consideration, but if so with great care, since they influence both reuse and speed; their main importance is in revealing difficult conflicts in need for further research. A factor can be divided into smaller parts, which can then be categorized again, and again. However, this would lead to an overly fine-grained theory beyond our purposes.

4.5.3 Improvement areas

Our improvement taxonomy (Figure 17) recognizes *business, architecture, processes, organization, knowledge and communication* factors as improvement areas, and *budget and resource allocation* (BRA-B), etc. as specific areas. It is useful for prioritizing and communicating improvements. Classification of our factors with this taxonomy is given in our technical report [78].

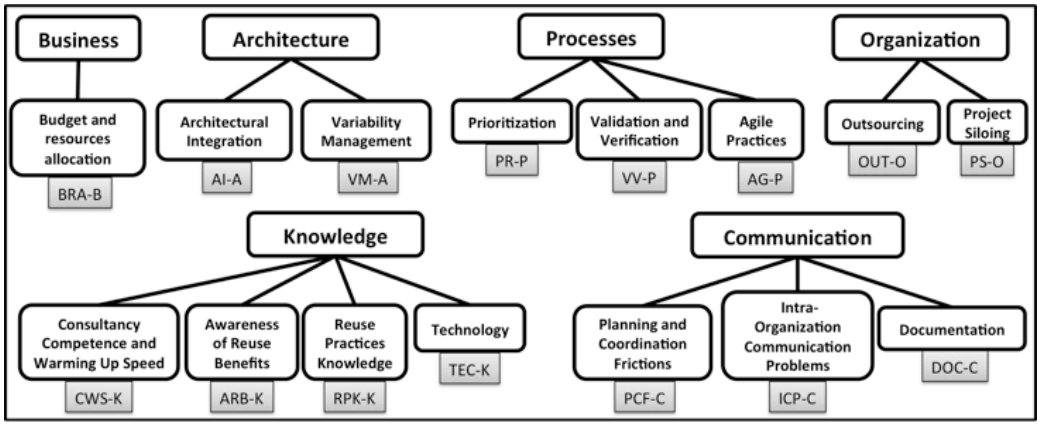


Figure 17. Improvement areas

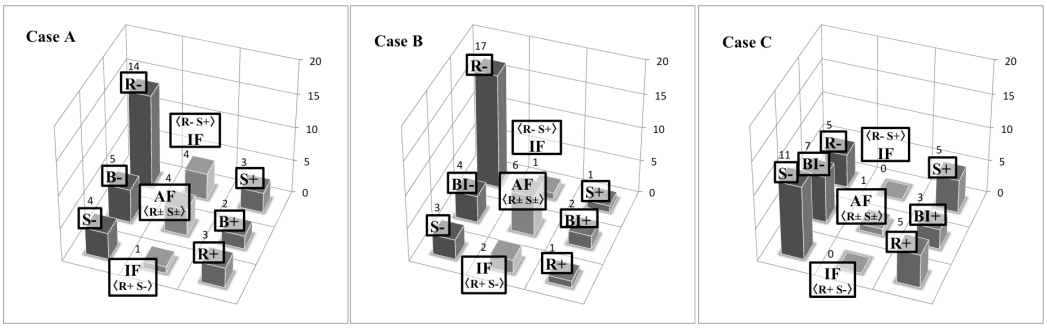
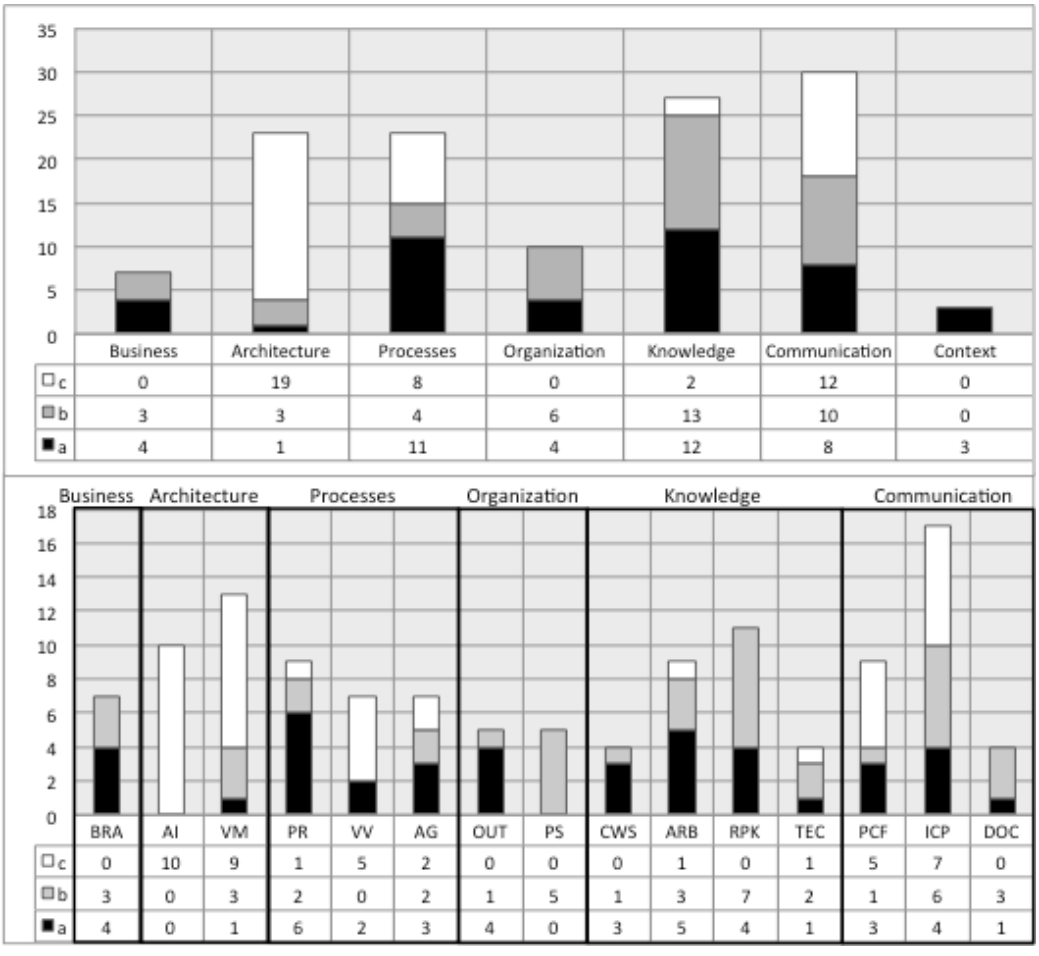


Figure 18. Distrubution of the factor on the map, specified by context



	Business	Architecture	Processes	Organization	Knowledge	Communication	Context
c	0	19	8	0	2	12	0
b	3	3	4	6	13	10	0
a	4	1	11	4	12	8	3

	BRA	AI	VM	PR	VV	AG	OUT	PS	CWS	ARB	RPK	TEC	PCF	ICP	DOC
c	0	10	9	1	5	2	0	0	0	1	0	1	5	7	0
b	3	0	3	2	0	2	1	5	1	3	7	2	1	6	3
a	4	0	1	6	2	3	4	0	3	5	4	1	3	4	1

Figure 19. Factors distribution with respect to the improvement areas and contexts

4.5.4 *Factor distribution*

The bar charts in Figure 18 show the distribution of the factors for each factor map (of case A, B, and C), whereas Figure 19 shows the distribution with respect to the improvement taxonomy; when present, we have included *context* factors. These diagrams suggest which areas the informants were most concerned with in each case and among all.

4.6 ANALYSIS – EXAMPLES FROM CASE A,B,C

4.6.1 *Case specific results*

The *factor maps* (in Table 7 and its underlying report [78]) and the distribution of the factors (Figure 18 and Figure 19) support the following inferences.

4.6.1.1 *Case A*

The black columns in Figure 19 show the number of factors found in each improvement area, for case A (in all 41 factors); the leftmost bar-chart in Figure 18 shows the distribution of these factors over our influence categories (*reuse inhibitors*, RI, etc.)

Knowledge is the most richly populated improvement area with 12 factors (see Figure 19); among its subcategories the richest is *ARB-awareness of reuse benefits* with 5 factors (all inhibitors). At a business and management level, where decisions about resource allocation, budget distribution and process tailoring are taken, it is crucial to have knowledge about the benefits of reuse (4 factors in *BRA – budget and resource allocation*). Developers and architects need *reuse practice knowledge* to implement reuse mechanisms (4 factors in *RPK – reuse practices knowledge*). Consultancy's lack of product specific knowledge hinders both reuse and speed (3 factors in *CWS – consultancy competence* factors).

Almost equally rich is the *processes* category with 11 factors, which are mostly connected to prioritizing the backlog (6 *PR-prioritization* factors). The product owner is responsible to achieve the optimal amount of reuse for cost benefits and improved replication speed and to maximize the delivered customer value for the 1st deployment speed through careful prioritization of tasks. When *"the same developers are working on both functionalities and architecture"* it brings benefits in terms of speed, whereas reuse benefits are unclear. In fact, developers' knowledge of both architecture and functionalities helps to recognize the common part to be reused. At the same time, a weak prioritization mechanism allows them to often prioritize customer functionalities to architectural improvements.

C*ommunication* is also a key area to be improved (with 8 factors), especially regarding intra-organizational communication (3 factors in *PCF-planning and coordination frictions* and 4 in *ICP – intra organization communication problems*). We have only one *documentation* factor, but it's a notable one. Writing a lot of documentation hinders speed (especially the *1st deployment speed*), while detailed information boosts assets' reuse. Documentation is also important to reduce long warming up periods of new employees and consultants. This factor is thus ambiguous.

In company A's *context,* the necessity to hit non-synchronized markets influenced negatively the implementation of *managed reuse*. The business goal of increasing the 1st deployment speed (also discussed for prioritization) drove the development. The effect is shown in Figure 18 (left): there are only 4 *S-* (indicating that the company is doing well with respect to speed) but 14 *R-* (indicating that the company is doing less well on reuse).

All 4 *organization* factors belong to the *outsourcing* subcategory, which is also a problematic approach (*AF* F-109 in [79]): the benefits of externally supplied components are strongly reduced (even nullified) by communication overheads and problems caused by attitudinal (cultural) incompatibilities.

The *good factors* in the map (the rightmost bars in Figure 18's leftmost chart) represent the interviewer's perception that *model-reuse* is an effective reuse practice, code centric Agile methods increase the *1st deployment speed* while code generation boosts both.

4.6.1.2 Case B

As for the factor analysis in case B, we refer to the grey bars in Figure 18 and the middle bar-chart in Figure 19.

We observe again (see case A) that most of the factors (13) are grouped under *knowledge*. The management and business parts of the organization don't take into account the benefits of reuse (3 *ARB-awareness of reuse benefits* factors) resulting in lack of budget dedicated to common assets development and architectural improvements (3 *BRA-budget and resource allocation* factors). This obstructs the implementation of *reuse practices* (7 factors in *RPK-reuse practice knowledge*), which is also represented by a substantial *R-* group of factors. However, an important context issue is the business model: having few and big projects and waiting for new requests by few special customers spread over time causes a high degree of uncertainty (see IF in Figure 18, middle), causing the risk of investing resources in core assets not valuable for the upcoming products.

Many factors belong to the *communication* between different units of the organization (6 *ICP-intra organizational communication problems* factors), in particular between developers and system engineers. An informant put emphasis on communication problems with the systemization, not being able to take reuse into account (factors in the *ARB-awareness of reuse benefits* subcategory that are *R-* in Table 7). Another informant, responsible for a different subsystem, stressed the same point, but described their capability to reuse a high percentage of the software through different projects thanks to continuous communication with system engineers. In both cases the agreement with system engineers influenced the success in reaching reuse.

As for *processes*, the standard structure of the projects hinders the introduction of agile practices (2 *AG-agile practices* factors) and the process customization for the small ones. The projects are also quite isolated, causing a phenomenon called "siloing" (5 factors in *PS-project siloing*): most of the decisions aim for the local optimum (see *AF* in Table 7), hindering the development of common assets among the projects (as many *R-* in Table 7 and [78] show).

We identified two reuse approaches in this case: the first one is the *ad-hoc reuse* of a light framework (of libraries), extracted from previous projects and expanded by developers every time it is reused in a new product. This is an *AF* (Figure 16); since this is a weak form of self-organized reuse (*ad-hoc reuse*) that is not managed, it's causing unhandled growth of artifacts that will need extra efforts to be managed, thwarting its benefits (the 7 factors in *RPK-reuse practices knowledge*). Thus, it's unclear whether this could be considered an advantageous form of reuse or not. The other approach is the *design-reuse* (complemented with quick reimplementation of code), which is profitable for small components in combination with well-explaining documentation.

The amount of documentation, as explained for the previous case, is a *complex* and *ambiguous factor* (see Figure 16) with many aspects to be considered.

4.6.1.3 Case C

For the third case (C) we refer to the white columns in Figure 19 and to the rightmost bar-chart in Figure 18.

This case is quite different from the others: a team is in charge of developing a shared component to be integrated in three products.

Most populated is *Architecture* with 19 factors. After a quick implementation phase, the component has suffered a delay due to problems concerning its integration into other products with their own architecture (10 *AI-architectural integration* factors). The lack of *VM-Variability Management* (9 factors) is directly connected with speed (see *S-* in Figure 16). Even though adaptors were correctly created to integrate the component into different platforms, well-implemented VM for handling the sets of functionalities delivered to the internal customers could have solved some coordination and communication problems.

Most of the *processes* factors (5) show that *VV-verification and validation* had a significant negative impact on speed (see *S-* in Figure 16). This suggests that the development of common assets should not be isolated from the test team.

As for the 12 *communication* factors, 7 *ICP-intra organizational communication problem* factors show that there were communication barriers between the units developing the products (that had to receive the shared component) and the unit implementing it. Some stable products were forced to integrate the common component, replacing the exiting functionalities: this caused *planning and coordination frictions* (5 *PCF* factors). *Communication* barriers with system engineers caused a long decision time before the start of the component development.

In conclusion, reuse-aware decisions taken at management and business levels (no *business* inhibitors mentioned, but one *R+*) led to successful achievement of *component reuse*. It's difficult to tell if the attempt was cost effective, especially in terms of speed (see the presence of as many as 11 *S-*). However, contextualized into a well-defined, long-term strategy of component-oriented architecture (see *R+* in Figure 16), this approach gains a higher value.

4.7 DISCUSSIONS

4.7.1 Generalization of output

Comparisons of our three cases are shown in Figure 19 and Figure 20.

From the even distribution of the 30 factors in the *communication* column of Figure 19, (12, 10, 8 for case A, B, and C), we can infer that communication is a common concern at all sites. Communication between teams involved with a common asset (for example when applying *managed component-* or *platform-reuse*) should be well supported, as should communication across projects related to the same product (17 factors evenly distributed in the *ICP-intra organizational communication problems* column). Good communication infrastructure is also necessary among the development teams, between teams, and the units these depend on, e.g. systemization and business units. (This is visible in the *PCF-planning and coordination friction* factors and the *budget and resource allocation factor*s). Without adequate communication, reuse risks being unorganized and ineffective (*ad-hoc reuse*), or it will cause a huge penalty in terms of speed (see the 11 factors belonging to *RPK-reuse practices knowledge* most of which are inhibitors).

Agile practices were discussed in all the cases (2, 2, 3 factors for A, B, and C), and most of the interviewers were very positive, considering them as essential to improve

agility on the market, customer satisfaction and 1st deployment speed (almost all of them are *S+*).

The implementation of an extensive form of reuse (*big component-, subsystem-, platform-reuse*) requires awareness of benefits, and an *agreement of purposes,* among all actors involved in product development: the business side, the systems engineering unit, and the developers. This suggestion comes from Figure 19 (9 factors in *ARB-awareness of reuse benefits*): the absence of inhibitors among the *ARB-awareness of reuse benefits* factors for case C (the only ARB-factor is *R+*, see Table 7) and the presence of inhibitors for cases A and B together with the successful implementation of *component-reuse* for case C, confirms that *ARB* factors directly influence reuse quality.

The *component-reuse* strategy of case C explains the prevalence of *architectural* (Figure 19) topics. Case A couldn't reuse the middleware for different projects, so no architectural issues were raised; case B hasn't experienced the development of a common component for different running products, which was the source of *AI-Architectural Integration* problems for case C. As for *VM-Variability Management*, it needs to be handled (case C) when possible to avoid product coordination overhead, while it seems to be particularly problematic in presence of high uncertainty about markets.

As shown in Figure 20, the influence *R-* is the largest category with 36 factors; these are mainly due to companies A and B focusing on the *1st deployment speed*. However, factors are not due to implementation of agile practices, but rather a lack of focus on the implementations of reuse practices.

The18 *S-* and 16 *BI-* show that many factors influence speed negatively; these are spread on many categories, even though *communication* and *knowledge* are the dominating ones.

The 19 *complex factors* (*IF1, AF, IF2*) indicate topics that need further in depth investigation, such as *the right balance of documentation*, *desirable tradeoff in prioritization,* and *(non) benefits of outsourcing.*

4.7.2 Limitations and threats to validity

Our research design is sensitive to following sources of errors, many of which are intrinsic to interpretive, case study research: (e_1) the *factor maps* are based on the study of four teams (plus three considered as secondary evidences) in three organizations, thus the set of factors is incomplete; (e_2) sampling is restricted in case C, which does not include project- and line-managers; (e_3) the relevance of the factors are influenced by the informants' daily work and daily concerns, some factors may have been overlooked; (e_4) the factors are influenced by the informants' retrospective reconstruction; (e_5) factor identification is influenced by the analysts' conceptions; (e_6) sampling and interpretation may be consciously or unconsciously biased to researcher concerns; (e_7) validation may not reveal and correct all misinterpretations and misclassifications made by the analysts; (e_8) statistics are based on the sheer number of factors, without weights; (e_9) our generalization is analytical, and may overlook significant differences in the complex contexts of our three cases.

The following precautions have been taken to reduce the effects of these sources: to handle e_1 we have chosen informants with long experience in the companies, with architectural roles and/or with management responsibilities. Moreover, we discussed the results in a workshop (see below); for e_2, we used open questions covering many perspectives to reveal factors beyond daily use; for (e_{4-5}), we have used a qualitative data analysis tool that supports traceability of factors to underlying data sources, presented results to employees of the same companies (in a workshop and draft papers), and taken the feedback into account; to handle e_{6-7}, we have used two

complementary kinds of validation: individual interview for case C, a workshop and (in progress) a survey to validate and deepen some of the factors; to address e_8 we have discussed also relevant factors belonging to categories not richly populated; to handle e_9, researchers have modeled and compared the contexts in search for differences that may influence the result.

The *factor maps* have been validated through a workshop. About 30 employees (with different roles) from the studied organizations were divided into six heterogeneous groups of 5-6 to discuss the maps for one hour and to present their views in a flipchart presentation. None of them questioned the right place of the factors on the maps. One group in particular expressed explicitly the correctness of the maps. Another group stated that "we recognize the many factors from case B too". Thus, we had a positive validation feedback on the correctness of our maps, but not on their completeness.

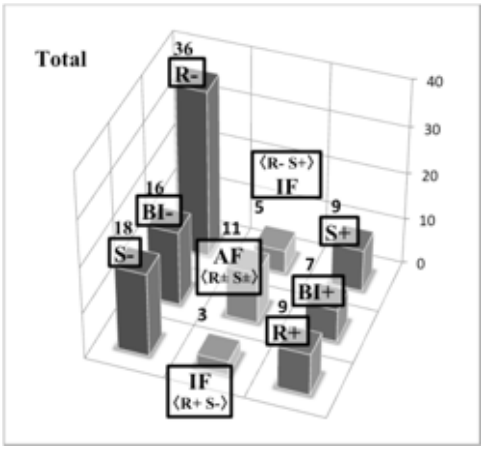


Figure 20. Total distribution of the factors

4.8 Related Work

Reuse and speed have been studied independently for years. Recently the scientific community focused on the combination of Agile and reuse practices and their influence on speed. Below are a few works that are particularly relevant to our topic.

Peterson presents a list of critical factors and their costs for the transition to an SPL [80]. Factors are categorized into *Cost Elements*, *Cost Drivers* and *Time Drivers,* but they are more abstract than ours and there is no notion of different kinds of reuse and speed. For example, for each area such as *architecture*, *processes*, etc. he enumerates different steps needed to obtain an SPL, such as *Domain Analysis*, *Current Architecture*, etc. Our factors belong to the same areas but they are more concrete. A good example of this is F-53: *Layer architecture against functional domain of the component caused delays in verification and validation of the component against all the layers* (detailed architectural problem). Further, Peterson, for each factor, estimates a cost. In our framework (see Figure 13) this covers the influence arrows connecting *reuse* and *speed* to *ROI of R&D* making the two papers complementary. The analysis in Peterson does not include the relationships among the factors. Finally, the aim of our case study was to discover the challenges of combining different degrees of reuse within an agile organizational setting, rather than focusing on the goal of having a completely established SPL.

Schmidt [81] identifies factors involved in the economic evaluation of a Software Product Line. In his *First Order Theory*, a set of constraints (development constraints, such as personnel resources, and process constraints) and attributes (software and market attributes) are listed and compared in order to put a cost emphasis on the

tradeoff between them. Reuse over time is considered at an abstract level: Schmidt states that time-to-market depends on resources rather than on reuse effort; he also relates time-to-market with revenues in different kinds of market. The paper is cost focused; it does not explain how factors are interrelated. No details are given, whereas our relationship analysis comprises a very concrete factor investigation.

Tracz [82], like us, mentions different kinds of reuse when explaining "Lesson #3: You need to Reuse More Than Just Code". The speed aspect is not discussed. We can see how some generalized conclusions drawn in our case study, like the importance of managerial decisions confirm Tracz's reuse advice.

Gaffney and Cruickshank [83] provide an economic model of software reuse. Different kinds of reuse are mentioned, and some examples of reuse application are described (ad hoc reuse, systematic reuse, domain engineering or application engineering reuse). The benefits of reuse are shown in terms of costs and productivity. However, speed is not included in their framework.

Morisio et al. [84] conducted a questionnaire with 32 projects to find success and failure factors in software reuse. Some of our *R-* and *R+*, are close to the ones listed in [84]mo. *Change nonreuse-specific processes* matches our conclusion that management, systemization, design, requirements, etc. have to be aware of the benefits of reuse and therefore willing to apply a suitable strategy (*ARB-K*). Addressing human factors, appears also in our conclusions, for instance in the need of a prioritized backlog to guide teams towards reuse. However, reuse is not related to speed.

Menzies and Stefanos work [85] is based on the same data collected by Morisio et al. [84], but it uses a different method of analysis. Menzies and Di Stefano question the conclusion drawn in [84] about the relevance of the top management commitment as an influencing variable. This slightly weakens the match between our results and theirs concerning the *R-* and *enablers*. However, as in the previous paper [84], speed is not taken into consideration.

Lynex and Layzell [86] propose a literature survey to identify reuse inhibitors. Some of our results are analogous, as we also identify *R-*. For example, in the organizations structure inhibitor, cited in [86], *No coordination of development efforts between overlapping areas* and in economics *Project managers have tight budgets that do not allow for extra investment in developing reusable components* can be mapped to our *PS-O* and *BRA-B* area of *R-*. However, as the name "reuse inhibitors" suggests, the paper [86] covers only a part of the problem, whilst our work includes some additional factors, as *R+*, *S-* and *S+* and other *CF*.

Diaz et al [72] use a systematic literature review to answer several research questions about the combination of ASD and SPL. They assert that principles of the two areas seem not to contradict each other (confirmed by our conclusions), although this is done only with abstract principles. Domain Engineering (where reuse plays an important role) is the most difficult part to conciliate ASD and SPL. Despite sporadic studies [87], the challenges in DE still have to be addressed, and our work deals with it.

Kakarontzas' approach [87] is based on elastic components and TDD is proposed to handle variability and to ensure the quality of the variants. Although the approach seems promising from the architectural point of view, the effectiveness is not yet supported by empirical evidences.

John D. McGregor [74] explains how SPL and ASD don't have to be necessarily separated. He proposes *method engineering* to tailor the fitting agile practice for the right area. However, he mentions *Micro Approaches*, but without specifying a concrete application in the matter of reuse. Here our study may prove useful. As a *Macro Approaches*, he states that agile teams can be used to develop core assets and that

interfaces between teams have to be minimized, which is confirmed by the *ICP-C* factors.

Some output from the XP 2009 workshop [88] seem to confirm our results, such as the necessity of avoiding barriers among units (*ICP-C* in our work), while documentation proved to be a complex topic of discussion (*AF* factors belonging to *DOC-C*).

On the basis of a literature review on ASD, Turk, France and Rumpe, give their perception about what the limitations of the ASD are [89]. They mention *Limited support for building reusable artifacts*, describing the importance of reuse practices, which bring especially long-term benefits. However, the authors don't address the issue. Our work focuses on this problem, trying to provide a first set of evidences.

In [50] a survey has been conducted with the aim of finding success factors in agile software development. Our case study confirms the importance of *Correct integration testing*, *Managers' style* (within teams) and *Project management processes* concerning the speed aspect. The reuse problem has not been mentioned in [50], but some success factors, for instance the *Right amount of documentation*, belongs to our *CF* because of the relationship with the reuse aspect. This suggests the value of our work in identifying sensitive points in combining Agile and reuse.

Hanssen and Fægri [73] describe, through a single case study, the successful combination of ASD (Evo) and SPL. The results are focused on applying the two approaches on different levels (*Strategic*, *Tactical* and *Operational*). Reuse is mentioned as a critical point in the combination. In the studied product, common components are reused and customer variability is managed. However, the authors don't focus on how these processes were integrated in the Agile setting. Besides, challenges in a medium size company delivering Internet based services are very different from the ones of large scale embedded software companies.

Kettunen and Laanti [90] provide a framework, based on industrial experience, aimed to understand how and why agility could be utilized for software process improvement (SPI) in large-scale embedded software product development. The framework explains how agility enablers support means, which are utilized to reach the goals. Our work focuses on concrete enablers (and inhibitors) and their relationships as responsible for goals such as speed and reuse qualities, when combining agile and reuse practices (not discussed in 19). In fact, as the authors themselves say, means and goals are based on enablers, which are of utmost importance. Some of the enablers cited by the authors are confirmed by our results. Some matching examples are *Do they [project team] communicate frequently and with good will? What dependencies do the product requirements have to external organizations? How easily information passes from one part of the organization to another?* with *ICP-C, PCF-C*, *PR-P* factors.

Petersen and Wohlin [75] conducted a case study in a large-scale development company to identify problems and benefits of incremental and agile methods. The work has analogies with ours, such as the case study method, the subject and the focus on enablers and inhibitors of speed. However, we conducted fewer and longer interviews in more companies, focusing also on the reuse aspect in relation to agile methods, collecting a broader set of interesting factors. Nevertheless, some of the results overlap: for example, the issues CI01 and CI11 can be mapped to F-44-48, and F-56-58, while advantages such as CA04 to F-84-87. From a more general point of view, our categories *VV-P*, *CFP-C* and *AG-P* have also been showed to be critical in [75].

4.9 CONCLUSIONS

Increased reuse and increased speed are common business goals in large-scale software development enterprises. For this reason it is important to understand their

relationships: such understanding benefits companies choosing among strategies for R&D, which need to make sensible tradeoffs. We explored this problem by conducting an exploratory multiple-case empirical study in three large-size Swedish companies involved with software development for embedded systems. Our purpose was to gather informants' experiences and perceptions in order to answer the following research questions:

Q_1: Which factors influence reuse?

Q_2: Which factors influence speed?

Q_3: Which factors influence both?

Q_4: Are influences positive, negative, or both?

Q_5: How can we decide which factors to address when implementing agile product development?

Q_1-Q_4 have been addressed by creating a *factor map* (Table 7), one for each case. The 114 factors placed here have been selected as the ones that influence reuse or speed or both. The *factor maps* showing the factors influence, outlined in Table 8. For Q_5, we have provided a distribution of the factors across different areas (Figure 19). The areas that were most populated are the ones most important to address.

Considering all cases, we have drawn the following conclusions:

- *Knowledge* deserves particular attention, especially *awareness of reuse benefits* (*ARB* factors) across all parts of the organization, which influences *business* decisions, *organization* structure of projects (*PS* factors) and the implementation and management of correct reuse practices (factors in *RPK*).
- Effective *communication* among organizational units (see *ICP* factors) and between different levels in the organizational hierarchy (*PCF*) is of utmost importance.
- Attention to *architecture*, especially integration of components (*AI*) and *variability management* (*VM*) are crucial for effective reuse without hindering speed.
- Similarly, early and continuous *verification and validation* (*VV*) *processes* are also crucial for speed.
- *Prioritization* (*PR, processes*) *documentation* (*DOC, communication*) and effective *outsourcing* (*OUT, organization*) are complex practices and need to be further studied before addressed in improvement projects.
- Agile practices (*AGP, processes*) are recognized as a key factor to improve the *1st deployment speed*.
- Reuse practices (especially for *managed reuse*) are necessary to obtain high *replication speeds*.
- Agile and reuse practices do not (directly) hinder each other: the efficient implementation of reuse practices is influenced by many other factors; Agile methods can be successfully implemented at a unit level, while implementing *component-reuse*.
- However, intra-organizational decision and communication issues (often due to reuse practices) may decrease the speed advantages gained by Agile Development.

Table 8. Influneces and answers to the RQs

Category of Factors *(Q_4)*	**Tot**	**A**	**B**	**C**	**Speed Infl.**	**Reuse Infl.**
Reuse Inhibitors (R-)	36	14	17	5		X
Speed Inhibitors (S-)	18	4	3	11	X	
Bi-Inhibitors (R- S-)	16	5	4	7	X	X
Reuse Enablers (R+)	9	3	1	5		X
Speed Enablers (S+)	9	3	1	5	X	
Bi-Enablers (R+ S+)	7	2	2	3	X	X
Inverse Factors (R+ S-)	3	1	2	-	X	X
Inverse Factors (R- S+)	5	4	1	-	X	X
Reuse En. & Inh (R+)	4	-	3	1		X
Speed En. & Inh. (R+)	1	-	1	-	X	
Other Ambiguous F. ($R_{\pm}$ $S_{\pm}$)	6	4	2	-	X	X
TOT (Q_3)	**114**	**40**	**37**	**37**		
Factors infl. reuse (Q_1)	**86**	**33**	**32**	**21**		
Factors infl. speed (Q_2)	**65**	**23**	**16**	**26**		

These conclusions, together with the *factor maps*, can be used as a guide for practitioners from contexts similar to the cases studied, especially for those working with agile product development. *Factor maps* support process improvement work by highlighting *good factors*, *harmful factors* and *complex factors* (Table 6) in an accessible way.

Recently, the focus of the scientific community and of practitioners has been directed towards combining reuse and Agile practices. Our work represents a step forward towards the understanding and the modeling of the interrelations between those and speed, an intricate web of interaction of business, process, architecture, and organizational phenomena influenced by knowledge and communication variables.

In our future works, we intend to focus on improving inhibitors in the most richly populated categories. In need of further investigation are the *inverse factors* (*IF*), to understand how much they contribute to one, and hinder the other aspect, and on *ambiguous factors* (*AF*), which involve the same aspect for conflicting reasons. As shown in the conceptual framework with the dashed arrows, the influence of the companies' contexts on the *manageable factors* and of the *reuse* and *speed* qualities on the *ROI of R&D* deserves a special attention (we are currently working on a model for the studied contexts).

5 INTER-TEAM INTERACTION CHALLENGES AND RECOMMENDATIONS FOR AMBIDEXTERITY

In order to achieve a successful business, large software companies employ Agile Software Development to be fast and responsive in addressing customer needs. However, a large number of small, independent and fast teams suffer from excessive inter-team interactions, which may lead to paralysis. In this chapter we provide a framework to understand how such interactions affect business goals dependent on speed. We detect factors causing observable interaction effects that generate speed waste. By combining data and literature, we provide recommendations to manage such factors, complementing current Agile practices so that they can be adapted in large software organizations.

This chapter has been published as:

Martini, A., Pareto, L. & Bosch, J., 2013 *"Improving Businesses Success by Managing Interactions among Agile Teams in Large Organizations"*, published in the proceeding for 4th international conference in software business (ICSOB 2013) [91]

5.1 INTRODUCTION

Large software industries strive to make their development processes fast and more responsive with respect to customer needs, minimizing the time between the identification of a customer need and the delivery of a solution. An open issue is how to scale Agile Software Development (ASD) from successful small software projects [15] to large software companies. One successful approach is to split the products in components and features and to parallelize the development using small, fast teams [10]. However, such approach brings the drawback that a team requires interaction with many other teams [16]. The support for such interactions with a high number of teams or with the surrounding organization in Agile methods is weak and not well explored [92]. Some studies highlighted how interaction issues often cause inefficiencies [93], [14],[94],[71], and hinder the speed benefits gained by the parallelization of the development [94]. Also, delays in interaction due to synchronization may turn fast individual teams into slow and frustrated teams constantly forced to wait for others, hindering the fast release of features [10].

There may be several reasons why the teams lose time to interact and to carry out tasks related to such interaction. This speed waste decreases their *interaction speed* and therefore their overall speed.

The purpose of this study is to identify the drawbacks of ASD employed in large-scale software companies related to interaction speed and their impact on business goals depending on speed.

The research questions addressed in this paper are the following: in the context of large scale ASD,

RQ1 What is inter-team interaction speed?

RQ2 How does inter-team interaction speed affect software business?

RQ3 What factors influence negatively inter-team interaction speed?

RQ4 How can a practitioner detect and manage such factors to increase inter-team interaction speed?

The paper investigates these questions through a multiple-case case-study with three software companies employing large scale ASD. We conducted exploratory group interviews, followed by qualitative data analysis, and member checking sessions.

Our contributions are:

- We define a notion of *interaction speed* as an externally visible property of *organizational boundaries*.
- We describe the *impact* of interaction speed on *business goals* dependent on *speed*.
- We identify factors associated with ASD that cause effects with negative influence on inter-team interaction speed.
- We provide recommendations based on the interviews and the exiting literature.

5.2 Literature Review

When surveying the literature, we have found a constant dilemma between, on the one hand, the need to create fast and independent Agile teams [10]and, on the other hand, the need to increase inter-team communications [95]. We found important to understand what interactions are affecting speed and what the causes are. Researchers in Global Software Development have studied interaction problems with the focus on geographically distributed teams [92],[95],[96],[16]. Recommendations found in [95] (*Optimally Splitting Work across Sites, Increasing Communication, Finding Experts, Awareness*) have shown to be important also in large organizations that are considered co-located. This suggests that in large software organizations, even if co-located, the size of the project creates some of the effects as the geographically distributed teams. Therefore, our research may be of value for GSD and vice-versa. In [97] the authors studies how knowledge management affects the coordination of teams.

A critical characteristic of ASD in interactions is the informal communication: it has been considered of value for managing volatile requirements, which makes the development flexible but creates challenges for inter-team communication and coordination [98]. In [96] informal communication is suggested as working well for XP with a strong bridgehead between the teams.

A socio-technical framework for evaluating technical and work dependencies has been studied in [99]. However, such framework heavily relies on artifacts representing the ongoing interactions among the employees, which requires reliable artifacts. Such artifacts are not available and are not representative in an informal ASD environment. Thus we found important to understand the interactions in such an environment.

Most of the abovementioned works study only parts of the problem from several perspectives. In [94] the focus is similar to ours, but the studied impact of Agile practices in communication is not related to speed and business goals. Our framework provides a set of factors-effects affecting speed, and recommendations to handle such factors that we haven't found in literature. The research in social psychology presents an opposite perspective on speed and interactions, in which time boundaries are claimed to influence group performance and interaction process [100].

5.3 Theoretical Framework

Our theoretical framework is based on an initial a priori framework, which has been continuously updated during the study and the analysis of the data [77].

We define *speed* (borrowing the concept from kinematics) as the amount of the *delivered value* (DV) divided by the *value delivery time* (VDT): the time between the perception of a need and delivery of value by some external party. (See Figure 21).

VDT is divisible into the time to identify the party (t_N), time until call for action (t_A), the time to commitment (t_C), and the time to delivery of value by the external party (t_D). We recognize the special case of *end-to-end speed*, where the need is perceived by a customer and the value delivered by a supplier.

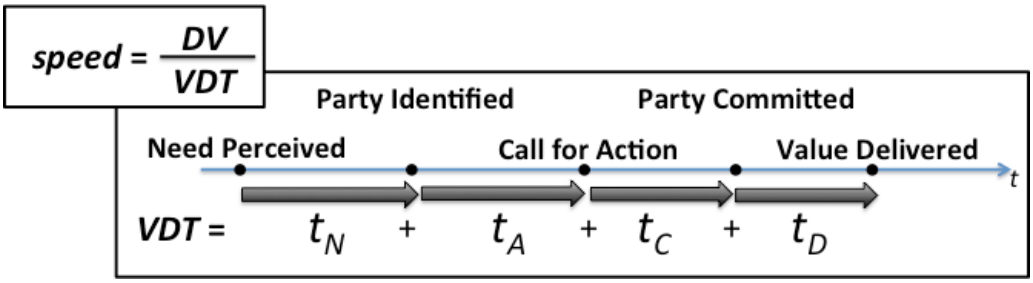


Figure 21 The definition of speed

A company that seeks to optimize its return of investment of R&D (ROI of R&D) must manage three end-to-end speeds (Figure 22). The speed with which customer needs lead to new product offers (*1st Deployment speed*), the speed with which new features are replicated in new products (*Replication speed*), and the speed with which change request to an existing product are realized (*Evolution speed*).

End-to-end speeds, in turn, depend on *interaction speed*: how fast teams (or other organizational units), resolve each others' needs. (Figure 23).

Interaction speed relates to both *inter-organizational interaction* (e.g., between teams at the customers and the client side) and *intra organizational interactions* (e.g., between a product management team and a team in a design unit). Each interaction involves several sub-interactions that address sub-needs, and also third party teams.

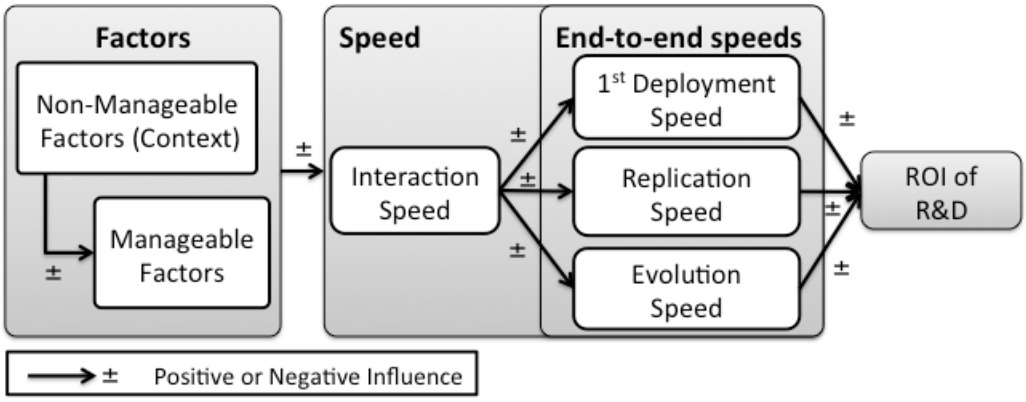


Figure 22 Three kinds of end-to-end speed and their dependency on interaction speed.

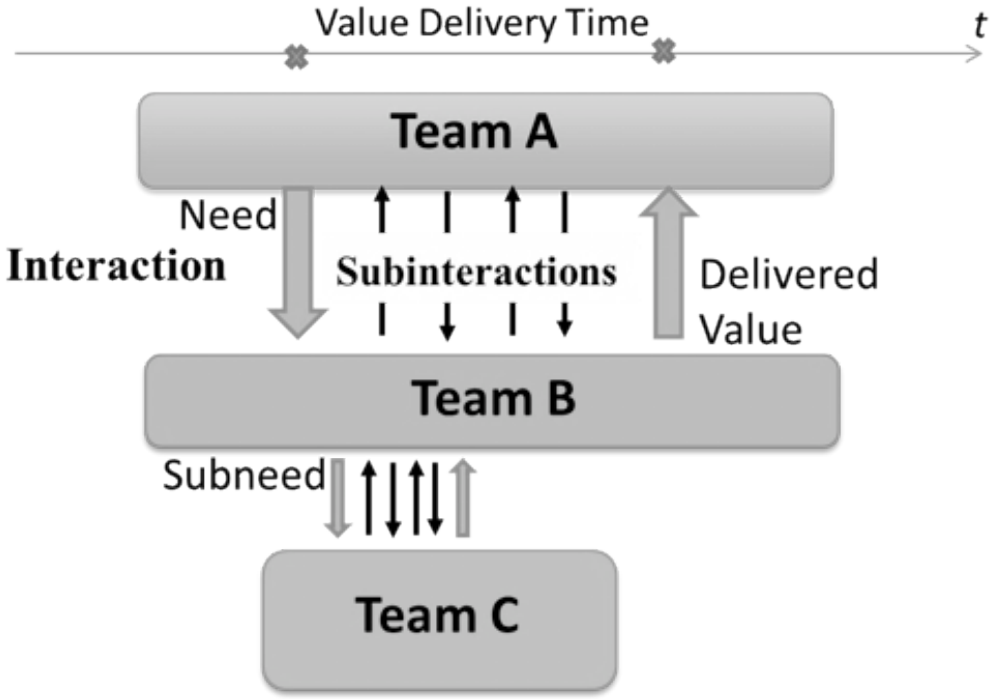


Figure 23 Interactions, sub-interactions, and interaction speed

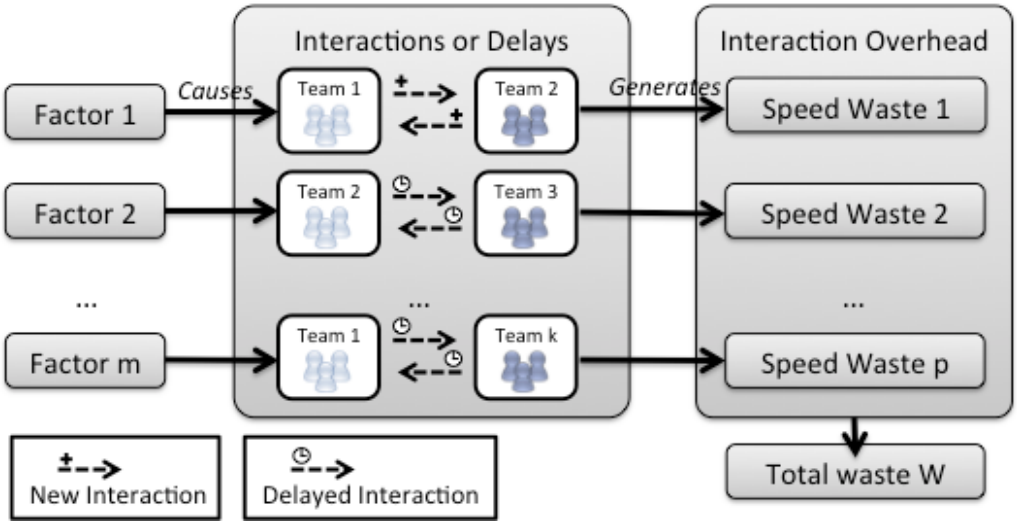


Figure 24 Factors creating or influencing interactions generating Speed Waste

Interaction speed depends on a number of organizational, architectural, and individual factors that may or may not be managed (Figure 22). To optimize ROI of R&D we must 1) understand what these factors are, and 2) find strategies to manage them.

In this paper we focus on the impact of interactions on VDT (we assume to have a fixed DV to be delivered, i.e. a set of features). We say that a factor generates *speed waste* if it is either *causing* or *delaying* interaction, which in turn increments VDT decreasing speed (Figure 24). The *total speed waste* (W) is the sum of all such speed wastes.

5.4 RESEARCH DESIGN

We planned a multiple-case case study with engineers and managers in product developing organizations. Our unit of analysis is the *cross functional agile team* and the phenomena of interest the *interaction* speed from the perspective of such teams.

Case Selection: To bring out the complexities of interaction, organizations were to be product-developing companies, with significant maintenance activities, and at least 100 developers. The companies studied were to have several years of experience of ASD. The cases chosen were three large companies with extensive in-house embedded software development. All were situated in the same geographical area (Sweden), but they were active on different international markets. For confidentiality reasons, we will call the companies A, B and C.

Case Description: Company A was a manufacturer of telecommunication systems product lines. The customers receive a platform and pay to unlock new features. The organization was split in cross-functional teams, most of which with feature development roles. Some of the teams had special supporting roles (technology, knowledge, architecture, ect.). Most of the teams used 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 before the feature was deployed.

Company B was a manufacturer of utility vehicles; the team developed a communication subsystem for one of their product lines. In this environment, the teams were partially implementing ASD (Scrum). Some competences were separated, e.g. System Engineers sat separately. Special customers requesting special features drove the business, and speed was important for the business goals of this company.

Company C was involved in the automotive industry. Some of the development was done by suppliers, some by in-house teams following Scrum. The surrounding organization was following a stage-gate releasing model. The team we studied developed in-house software, served some projects with different releasing deadlines.

Data collection: data collection was structured in three phases: initial workshops with participants from A, B and C; focus group meetings; validation sessions for reviewing the results.

In the first phase, we conducted semi-structured group interviews with team members (developers and architects), line managers and process specialists. Interviews included participants with mixed roles and revolved around Figures 1-3.

In the second phase we ran 3 focus groups, one for each company. We studied the phenomenon from the team perspective. We included senior developers, team leaders, architects and testers. In this phase we focused on extracting the main factors that were causing or influencing interaction speed. We ran the focus groups separately for each company. We discussed the problem by using models 1-3, then we asked the participants for situations in which the team was suffering from interaction and finally we injected the information from the previous sessions.

In the third phase, after the data analysis, we ran an interview session for each company for validation purposes. Some of the same participants were involved in this process, to adjust researcher's representation of the data. Finally, a short validation workshop with 2 employees from all the companies was conducted.

Data Analysis: After each session we analyzed the recorded interviews to develop models and first results to be discussed in the following sessions. The analysis of the data was carried out between the phase 2 and 3 and also afterwards, to refine the results. We inductively further developed the initial theoretical frameworks and we populated them with factors, effects and improvement practices emerging from the

data. We defined each factor, classified it as either *generating* or *influencing* interaction, classified by polarity, and illustrated the importance of managing the factor to increase interaction and end-to-end speed. We have also extracted some suggestions for improvement practices, although such hypothesis need further research.

Synthesis: On the basis of this analysis and suggestions by the informants in interview data, we formulated mitigation strategies for the factors found. Such factors and the mitigation strategies were reported back to informants, and the feedback recorded, analyzed and incorporated in our results.

5.5 FINDINGS

In the following we show the factors and the effects distilled from the analysis. We have identified 10 *Root Factors,* all manageable. Each factor produces one or more *interaction effects* that are observable in the company. Such effects (and the factors as well) have a negative influence on interaction speed. We have recognized 8 effects:

Table 9. Effects and their explanation

E1. Long waiting time to comm.	A team has to wait before communicating with other ones. This increases t_N, t_A or t_C (Figure 21).
E2. Long waiting time for value	As the previous one, but the team is waiting for the realization time t_D needed from the other team to deliver the value.
E3. Intense communication	Each instance of inter-team communications requires a long time. Again, this may influence t_N, t_A or t_C.
E4. Corrupted communication	The information received by the team is insufficient to deliver the requested value. This, in turn, may cause intense communication or high interaction frequency (see E5).
E5. High interaction frequency	The number of interactions between two teams is too high (i.e. it clearly hinders the focus on the current development). An instance of this effect occurs when a member in the team is continuously consulted for his or her knowledge by many other teams. This phenomenon has also been called "backpacking" in the interviews.
E6. High task frequency	A single interaction may require many tasks to be carried out in order to deliver the value.
E7. Heavy interaction tasks	The time t_D is long because of the large amount of time required for carrying out the task to deliver the value.
E8. Corrupted value	A (sub-) value has to be delivered to complete the interaction. However, the received value doesn't satisfy the need that started the interaction.

The *length* and *frequency* mentioned for characterizing the effects are not defined in detail because of the exploratory nature of the study: we have used "long" and "high" to emphasize what seems to be "too much" from the interviews. The same holds for "high frequency". We couldn't establish a specific threshold for the frequency: however, a higher number clearly corresponds to the increment of VDT.

In the following we list 10 Root Factors. For each, we give a definition and we explain what interaction effects they cause in terms of speed. We also explain how they are connected to ASD and what recommendations we suggest, based of the data and the literature.

Table 10. Root Factors

F1. Knowledge unavailability	If a team doesn't have all the knowledge to develop a feature independently, they will try to interact with an expert outside the team, creating interactions. The may have to wait for the expert to be available. The team may alternatively decide to make assumptions on the answers that lead to redo most of the work. The expertise may encompass different kinds of knowledge, such as domain, product architecture and technical knowledge [97]. This factor is connected to ASD and the trend of defining small and self-sufficient teams: the more independent they are, the more

	isolated, the less effective inter-team communications might be [101]. **Recommendation R1**: make available part-time experts serving different teams and covering critical knowledge (the most requested one). The idea is to decrease the workload in the actual team that is not related to the critical expertise from the expert and make him an inner consultant serving the other teams. Grouping interactions in a defined time-box would avoid high frequency of interactions. This involves a process of identifying the critical knowledge, allocating time to the expert broadcasting the information of such availability to the teams.
F2. Expert's reputation	If an employee has a high reputation of having a specific knowledge, the person will be contacted often. Reputation is not only based on the real knowledge of an employee, but rather on his or her social reputation. ASD principles value social interactions over formal knowledge, amplifying the effects of this factor on interaction speed (as also hypothesized in [95]). Thus, some experts might be more consulted than others because of their social status: this might unbalance the interactions among the teams.
F3. Unclear requirements	The team receives requirement specifications for the features. They may have two interaction problems: the long waiting time before the team is able to receive the specification, or the continuous interaction for clarification of the requirements afterwards. The two problems are connected, according to the interviewees: the time spent on the feature preparation determines the quality of the specification, which influences the elaboration time by the team. **Recommendation R2:** the time spent on creating requirements and architecture artifacts might be decreased in order to start the development as soon as possible: to counter balance this approach, part-time roles of architects and product owners should be established in order to provide constant support to the team during development, avoiding the continuous interaction for clarification.
F4. Unexpected Feature Dependencies	Two features may be designed to interact with each other through APIs or through a component. In some cases, dependencies pop up unexpectedly, e.g. due to indirect (software) interactions or because of socio-techincal reasons (as studied in [99]). The team needs to negotiate APIs or to frequently merge changes on a shared component. The dependencies problem is not covered by any known Agile practice. **Recommendation R3**: In this case, as in other kinds of team (see F3), a brigdehead between the two teams would help coordination. Face-to-face communication is infact beneficial as highlighted in [102]
F5. No co-location	Large organizations are forced to spread teams in space. According to our interviews, even the distance of one floor makes them distributed, with consequent delays and lack communication and commitment. **Recommendation R4**: The interviews suggest that the teams that have to interact more intensely should be located closer. It can be considered as another level of co-location with respect to intra-team co-location. Even if something like "inter-team co-location" is not mentioned by ASD per se, it can be considered an extended version of the intra-team co-location suggested by the Agile principles. There are also attempt in literature to mitigate this factor in GSD, e.g. [103],[104]
F6. Lack of common time	Teams may need to synchronize in meetings, which requires common available time. If a team decides to not allocate time for interaction or the allocated time-slots don't match, there is a lack of communication or long waiting times. Causes may be the different locations, different time zones (or with different slots of working hours), calendar interferences or low prioritized interaction. This has also been highlighted in [95]. **Recommendation R5:** some agile practices, such as SCRUM, include support for meetings between SCRUM masters. However, other kinds of programmed available time could be considered, e.g. as mentioned in [105]. **R6:** Also shared calendars would help provide better alignment [95].
F7. Mismatch of team's styles of	Different teams may have different "styles" of communication, which may cause delays: e.g. one team mainly uses e-mails and doesn't want to meet in person, whilst the other doesn't reply often to e-mails and is used to

communication	communicate through face-to-face meetings. The effect is a lack of communication. Another issue may be the different uses of knowledge containers such as boundary objects (e.g. wikis). The Agile culture of letting teams have their customized processes somehow encourages this mismatch. **Recommendation**: inter-team interfaces between team that need interactions should be improved, for example, with bridgeheads, employees having a strong influence in more than one team ([96] and **R3**). This could also affect the study and the composition of the teams: each team would require the presence of someone socially connected to another team that requires a lot of interactions. Also **R2** may be applied, if architecture or product management teams are involved. Some other practices can be found in communication literature, e.g. [102], [103]
F8. Slow resource indexing	When a member of a team needs to interact, he or she needs to find the correct person or team to interact with. The time spent on such activity (t_N, Figure 21) may be long and therefore delaying. The informality suggested in ASD seems to work as an amplifier for this factor. The choice of consulting people over formal documents creates "Backpacking" (see E5).
F9. Low prioritized interaction	Once an interaction is needed, the involved parts (single employees or whole teams) have to prioritize the interaction as an on-going task. If the interaction is considered as "low priority", the team will delay tasks and communication, hindering the other team(s) involved. **Recommendation R7**: Tools for creating awareness would help in the understanding the overall situation of the involved teams [95]. Again, the presence of people also connected to other teams would enhance commitment (**R3**, **R2**).
F10. Inter-personal conflicts	Two employees in different teams (or even the whole teams) may consider each other "enemies" (for personal or political reasons). Interactions between these employees may be strongly hindered by delays and corrupted information. Again, a work environment strongly built on social interactions may amplify this factor. Some recommendations for this factor have been suggested on different levels in [106]. However, some social aspects in software development and related (agreed) guidelines need further research [106],[107]. The authors in [108] also suggest that these conflicts may be rooted in unclear requirements (F7).

Finally, we can see, in table 3 below, which factor is responsible for which effect.

Table 11. Factors causing observable effects with negative influence on interaction speed.

Manageable Factors	Cause	Effects							
		E1 *Waiting comm.*	**E2** *Waiting value*	**E3** *Intense comm.*	**E4** *Corrupted comm.*	**E5** *High inter. freq.*	**E6** *High task freq.*	**E7** *Heavy tasks*	**E8** *Corrupted value*
F1 *Knowledge unavailability*	➔	x	x		x	x	x		x
F2 *Expert's reputation*				x		x		x	x
F3 *Unclear requirements*			x	x	x	x			x
F4 *Unex. Feat. Dependencies*				x		x	x	x	
F5 *No co-location*		x	x		x				
F6 *Lack of common time*		x							
F7 *Mismatch of comm. styles*		x		x	x				
F8 *Slow resource indexing*		x			x				
F9 *Low prior. interaction*		x	x		x				x
F10 *Inter-personal conflicts*		x	x		x				x

5.6 DISCUSSION

In this section we explain why our results are relevant for the software business, how the factors can be discovered and how recommendations can be applied by practitioners (managers or teams).

As explained in [71] and in Figure 22, there are three end-to-end speeds influencing Return of Investment: 1st deployment speed, replication speed and evolutions speed. Interaction speed affects all of them, as explained in the following paragraphs.

- *1st Deployment Speed:* when a set of features is released for the first time, the speed is affected by the interaction speed among the teams that have to integrate the features. This kind of speed helps *hitting the market fast* to anticipate the competitors. Fast deployment speed also *shortens the loop in market testing*.
- *Replication Speed:* when a feature is embedded in a previous release, interactions are needed between the team responsible for the new features and the teams that had developed the former ones. Replication increases ROI when the effort made for the 1st deployment speed is spread on the *release of new products and services based on the existing software.*
- *Evolution Speed:* when a feature needs to be changed after its release, such changes will affect other features, requiring interactions again. The speed in *reacting upon a change request* can be critical for gaining the trust of the customers.

Managers want to reach the abovementioned business goals. Delays over the schedules may be due to speed wasted in interactions. Managers may recognize it but they need the team(s) to observe the effects E1-E8. Since each effect is related to one or more root factors (F1-F10), managers can immediately investigate the status of such factors in the teams to find which one is the cause for the effect. In Table 2, the connections between factors and effects reduce the solution space for the investigating manager, who saves time and resources. In case the factors are recognized, both the team and the manager (depending on the factor) may decide to apply the recommendations (R1-R7). This process is outlined in Figure 25.

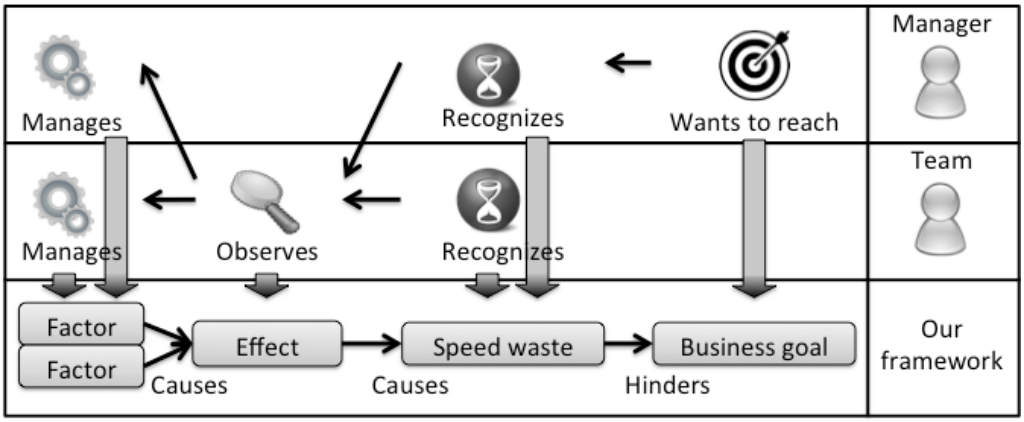


Figure 25 How the practitioner can use our results.

The results reveal that the employment of ASD in large organization brings new root factors hindering inter-team interaction speed and amplifies some of known ones that do not (or limitedly) appear in small projects. Practitioners should be aware of these factors when implementing ASD in large organizations. From the synthesis of the data, the comparison of the factors with Agile principles and solutions from literature, we have summarized a set of recommendations in form of guidelines, which need to be refined by further research, aimed at managing the factors (R1-R7). We have linked the recommendations with the factors (Table 2): this way, whenever an effect is visible, the

practitioner could check the corresponding factors and apply the given recommendations (or an adaptation to the company's current situation).

Agile practices (in small contexts) stress the importance of having the customer on site In large companies, the team has to follow requirements defined by a reference architecture and it has to negotiate various kinds of requirements with other teams, i.e. the team has more than one customer: product management, architects and the other teams. Following the "customer on site" principle, all these stakeholders should be available and provide fast feedback to the team. This can be achieved by the creation in the organization of part-time architects, product managers (as in [98]) (R2-R3) and social/formal bridgeheads among interacting teams. Critical expert knowledge should be identified and made available to the teams in form of part-time experts serving multiple teams. As mentioned in [95], a team needs to be aware of the other teams to align with them: shared calendars (R6) and other tools for awareness (R7) would help teams synchronizing. Another solution is mentioned in [105], where "communities of practice" are proposed as programmed events to bring together people from different teams and to spread knowledge (R5). In ASD also the effect of social networks and political dynamics may be amplified, so interacting teams have to be placed close or to mitigate conflicts [106], [92] (R4). Also, the social capital [109] of the organization has to be carefully developed and maintained (for example by defining competence models [110]) in order to increase interaction speed.

5.7 THREATS TO VALIDITY AND LIMITATIONS

In this section we list and explain the limitations for this study. The factors F1-F10 and their recommendations R1-R7 have been analyzed in terms of interaction speed. Other impacts have not been taken in consideration. The information is based on employees' statements and may be biased. The causality of the factors is a hypothesis, and it's not supported by quantitative data yet. The practices are hypotheses synthesized from interview and have been validated in the last session of interviews, but not with precise empirical measurements. One possible threat to validity is the evaluation apprehension: the employees were interviewed usually in groups, which helped balancing statements. To handle the mono-operation bias we collected data from three companies and in some cases from more the one site. As for background influence, we interviewed various roles, from managers to programmers. We limited the threats to conclusion validity (such as influence posed on the subjects) by injecting the preliminary results only after the respondents gave their statements. The threat to external validity (generalizability) has been limited (but not completely solved) by studying three cases with common attributes: size, development domain (i.e. embedded systems) and introducing ASD. We highlighted the differences in the section about their contexts.

5.8 CONCLUSIONS

The effective implementation of ASD in large companies developing embedded software may be the way for the successful achievement of business goals depending on speed. However, Agile teams need to avoid unnecessary interaction and, when unavoidable, to interact efficiently among them and with the rest of the organization. We have described interaction speed (**RQ1**) through the definition of a set of models to frame it with respect to large organizations employing ASD. The reduction of interaction speed negatively influences three business goals: 1st deployment speed, replication speed and evolution speed (**RQ2**). To increase interaction speed and therefore reach such goals, we provided the practitioners with effects (E1-E8, Table 1) observable in the organization, and the factors causing them (F1-F10, Table 2)(**RQ3**).

Finally we have proposed a set of recommendations (R1-R7) to manage such factors in order to complement the practices suggested by ASD (**RQ4**).

Future research includes further strategies for managing the factors and a quantitative study of the effects. The long-term objective may include a measurement system for connecting a quantifiable amount of speed waste with the effects and with specific indicators for the factors.

6 INTER-GROUP INTERACTION CHALLENGES AND MITIGATING SPANNING ACTIVITIES

The adoption of Agile Software Development in large companies is a recent phenomenon of great interest both for researchers and practitioners. Although intra-team interaction is well supported by established agile practices, the critical interaction between the agile team and other parts of the organization is still unexplored in literature. Such interactions slow down the development, hindering short-time and long-term responsiveness, which in turn hinders the achievement of ambidexterity.

We have employed a two-year-long multiple-case study, collecting data through interviews and a survey in three large companies developing embedded software. Through a combination of qualitative and quantitative analysis, we have found strong evidence that interaction challenges between the development team and other groups in the organization hinder ambidexterity and are widespread in the organizations.

This paper also identifies current practices in use at the studied companies and provides detailed guidelines for novel solutions in the investigated domain. Such practices are called boundary-spanning activities in information system research and coordination theory. We present a comparison between large embedded software companies employing agile and developing a line of products based on reused assets and agile companies developing pure software. We highlight specific contextual factors and areas where novel spanning activities are needed for mitigating the interaction challenges hindering speed.

This chapter has been accepted for publication as:

Martini, A., Pareto, L. & Bosch, J., 2014. *"A multiple case study on the inter-group interaction speed in large, embedded software companies employing Agile"* accepted in Journal of Software: Evolution and Process

6.1 INTRODUCTION

Large software industries strive to make their development processes fast and more responsive, minimizing the time between the identification of a customer need and the delivery of a solution. Agile Software Development (ASD) gives support for such goals, and has proven successful in small software projects in the last decade [6]. Less evidence can be found on its success in large software companies, where research is still ongoing [11]. The adaptation of ASD to embedded software, driven by an overall physical product development process [20] seems to be even more challenging and unexplored.

ASD stresses the importance of responsiveness, and the continuous interaction between actors involved in the software development process. In large software companies, an agile team needs to interact with a number of groups: the customers (or their surrogates, i.e. product owners) or other stakeholders, such as hardware development teams or system architects, suppliers [70], and other agile or non-agile teams [91]. Some studies highlight how interaction among several social groups is often a critical factor, hindering the speed benefits gained by the parallelization of software development [71], [93], [94], [111]. Specifically, several interaction challenges has been found to hinder three business goals dependent on speed: first deployment speed, replication speed and evolution speed [71].

However, there is a need for understanding which boundaries exist between the agile team and other groups with the surrounding organization and which interaction

challenges are more relevant [6], [92], in particular for the embedded software domain. Existing theories suggest the need for boundary-spanning activities in order to establish a coordination strategy [22], [52], [53]. Such activities are meant to fill the interaction gap existing between two social groups by spanning the existing boundary.

We employed a two-year-long case-study involving three firms to understand which critical interaction challenges hinder achieving business goals dependent on development speed and how widespread such challenges are in the companies. We have also inquired the current status of practice regarding spanning activities at the studied companies. This paper focuses especially on large embedded software development companies employing ASD. We hypothesize a significant difference between embedded and pure software cases reported in literature. In relation to context, we aim at answering the following research questions:

- RQ1 Which inter-group interaction challenges influence business goals dependent on speed in large-scale software development?
- RQ2 Which inter-group interaction challenges are more critical and widespread in the organizations?
- RQ3 Which spanning activities for mitigating the interaction challenges are currently employed or missing by practitioners?
- RQ4 What are the differences between embedded software companies and pure software companies in terms of interaction challenges and spanning activities?

This paper has three parts and the main contributions are:

- a list of interaction challenges hindering several business goals dependent on speed. The list is ranked based on the level of recognition (strong, controversial, weak) and the spread through the companies. The challenges also show critical organizational boundaries, and are discussed in relation to companies' contexts and participants;
- a set of spanning activities, objects and coordinator roles that are used by the participants or are missing. Such activities can be used for the creation or adaptation of a coordination strategy that will improve speed by mitigating the interaction challenges and therefore increasing speed;
- a comparison of our selected cases (embedded software domain) and the pure software cases analyzed in literature focused on the same topic. We show a significant difference in the contextual factors and the areas where spanning activities are needed.

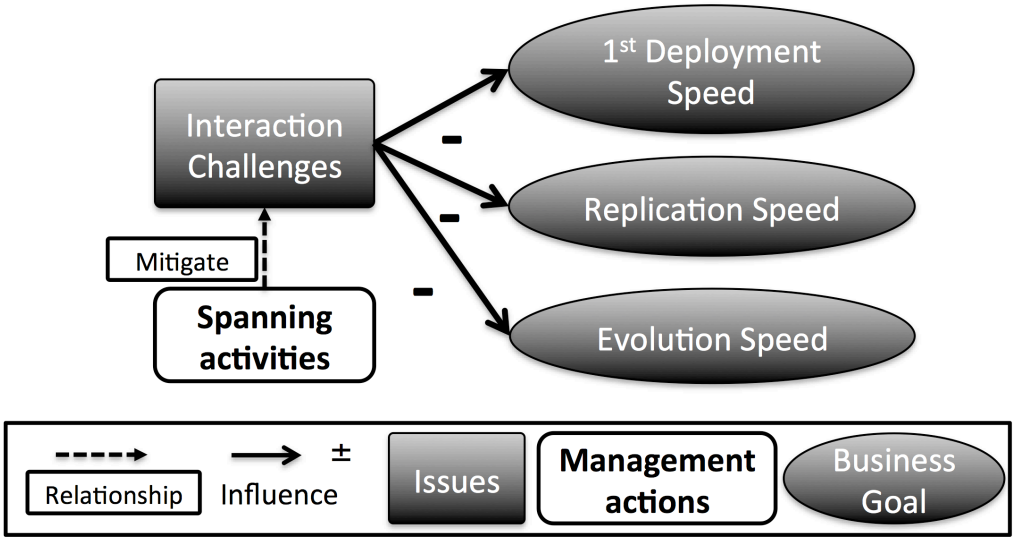
In the next section we describe our conceptual framework and present the theoretical components on which we built our research: speed, social interactions, spanning activities and ASD. Then we describe our methodology, a two years long multiple-case case-study including three firms, in which we have employed both qualitative and quantitative methods. We then present our results and their analysis, in which we show the previously mentioned contributions. The following section will discuss the generalization of the main results, how they inform the research questions and how they enrich the current body of knowledge in software engineering. The paper ends with the conclusions and our targets for future research.

6.2 BACKGROUND

Figure 26 shows our overall conceptual framework: in the observed software companies we found *interaction challenges* between the agile teams and other social groups. Such challenges hinder three business goals dependent on three kinds of speed: *first deployment speed, replication speed* and *evolution speed* [71] (shown by "-"

arrows in the picture). Such interaction challenges can be mitigated by the application of *spanning activities*. The results will be presented using this framework: the discovered interaction challenges, which goal they affect and which spanning activities are present or are missing in the observed companies.

Figure 26. Overall conceptual framework for managing inter-group interaction speed



6.2.1 *Speed*

Many studies dealing with speed, e.g. [71], [93], [94], [111]–[114], identified interactions among several actors in the software development process as a major generic influencing factor. Despite such recognition, we have found no studies that tackle this problem in depth and in relation with speed.

According to previous research results [71],[91], large companies developing multiple products with a substantial amount of embedded software need to manage three end-to-end speeds (Figure 26): the speed with which customer needs lead to new product offers (*first deployment speed*), the speed with which new features are replicated in new products (*replication speed*), and the speed with which change request to an existing product are realized (*evolution speed*). Lean Software Development also stresse the importance of end-to-end speed for the release of a product over the local speed of a single team, in order to optimize the whole instead of local output [115], [116]. Such principle differs between Lean and ASD, where instead the team speed is emphasized, which implies a distinction in companies' development approaches in practice [6], [17]. However, several mixed "Leagile" approaches have been applied according to the cases studied by [17], showing that the combination of principles is possible. For this reasons, we borrow the idea of waste elimination from Lean literature, where speed has been *defined* as "absence of waste" [115], [116]. Therefore we investigate the source of such speed waste, more specifically the speed waste due to interaction challenges (see next section) between the agile team and other parts of the organization hindering the previously mentioned end-to-end business goals. We also aim at finding possible solutions for such challenges: the results might be used to complement existing agile practices, used by the teams to achieve local speed, with activities that would benefit the end-to-end speed according to the Lean principles.

6.2.2 *Social interactions in Agile Software Development*

One important element in ASD is interaction, which also influences both the delivered value and the time spent on coordinating all the actors in a large organization.

The *development team* is a *social group* (*Group dynamics research and theory*, 1953) of interdependent employees responsible for the development of a portion of software, which might be a feature or a component. Other social groups may be a test unit, a supplier, a management team, etc. Between two social groups stands an *organizational boundary* [51]. *Interaction challenges* may occur between the development team and any other group in the organization. These challenges might hinder the achievement of speed or other business goals, as reported in several studies [71], [93], [94], [111]–[114]. However, the lack of in-depth studies on such interaction challenges, especially related to business goals dependent on speed, motivated our research questions:

RQ1 Which inter-group interaction challenges influence business goals dependent on speed in large scale software development?

RQ2 Which inter-group interaction challenges are more critical and spread?

6.2.3 Interaction challenges and spanning activities

In order to mitigate the interaction challenges, it's important to understand how the practitioners currently avoid such issues and therefore what software development practices are still missing. Such practices involving several social groups within an organization have been recently referred as spanning activities in information system research [52]. Such concept has been used by Strode et al. [22], who proposed the use of the current results from coordination theory [53] and analyzed how co-located agile projects coordinate. Among the coordination mechanisms, they mentioned spanning activities as: "*Boundary spanning occurs when someone within the project must interact with other organizations, or other business units, outside of the project to achieve project goals. There are three aspects to boundary spanning: boundary spanning activities, the production of boundary spanning artifacts, and coordinator roles*".

Such definition allows us to consider interactions challenges within the broader scope of project coordination: we take inspiration by Strode's framework for analyzing and organizing our results and for comparing our cases with the only reference in literature concerned, although only partially, with spanning activities.

Strode's framework proposes the definition and description of *spanning activities*, of *spanning objects* (artifacts) the *coordinators* (roles) responsible for carrying them out in the organization. Each activity needs to be carried out also at a specific frequency, i.e. *per-project*, *per-iteration* and *ad-hoc*. Our results might be used in the creation of an overall coordination strategy, which is, however, beyond the scope of this paper.

According to the previous definition and the findings highlighted by Strode et al.[22], Levina and Vaast [52] and Malone and Crowston [53], it's necessary to establish spanning activities for mitigating interaction challenges. This leads to our next research question:

RQ3 Which spanning activities for mitigating interaction challenges are currently employed or missing by practitioners?

6.2.4 Agile in large scale embedded software development

Embedded software is developed to control machines or devices that are not considered generic purpose computers. Such software is typically specialized for the particular hardware that it runs on and has to match specific time and memory constraints. Examples from such domains are cars, telephones, modems, robots, appliances, toys, security systems, pacemakers, televisions and set-top boxes, and digital watches.

ASD had a great impact on software development, and has been the subject of researchers' attention in recent years [6]. Companies developing software for specific domains, such as embedded software development, have shown a trend to adopt ASD [19], [20] However, such development process shows properties and difficulties that are not common to generic software development, and which have to be taken care of when employing ASD. For example agile teams depend on strict hardware requirements and resources and the necessity of following the overall product development [19]–[21].

Consequently, we make the hypothesis that the embedded software domain would bring more and/or different interaction challenges and therefore it would require different boundary-spanning activities. The aim is to answer RQ4:

RQ4 What are the differences between embedded software companies and pure software companies in terms of interaction challenges and spanning activities?

6.3 METHODOLOGY

In this section we describe our strategic and tactical research decisions: we conducted a multiple-case case-study as a tool for data collection and the combination of quantitative and qualitative data analysis, as recommended by Yin [62].

6.3.1 Conceptual framework

The components of our conceptual framework described in the previous section were continuously updated with inputs from empirical studies and literature reviews. We have combined inductive and deductive approaches for theory building, as described in [77] and as an essential feature of the Grounded Theory approach [55], [61].

6.3.2 Companies' contexts

The case selection followed a replication logic strategy, a tactic recommended in case study design [62], which can consist of either choosing cases that are similar and therefore likely to provide similar results (*literal replications*) or choosing cases that are dissimilar and therefore likely to provide contrasting results for predictable reasons (*theoretical replications*). The cases that we have selected represent literal replications and therefore we expected to produce results that were similar. On this regard, we chose all companies developing embedded software, and we aimed at strengthening the results by inquiring several sources (*source triangulation*) as recommended in [63]. Our cases can, however, be considered theoretical replications when highlighting the differences compared to the ones reported in literature: to fulfill this purpose, we will discuss our cases with the ones included in [22].

Our cases were three large product-developing companies (A,B and C), all with extensive in-house embedded software development. All were situated in the same geographical area (Sweden), and active on several international markets. Table 12 shows the context details for each of them. As for the Organization fields, it's important to notice that, although the actual organizations were large, including hundreds of employees, we needed to focus our research to units of analysis [62] that would make our investigation feasible.

Company A was involved in the automotive industry. Some components were developed by in-house teams following Scrum, whilst others were outsourced to suppliers. The surrounding organization was following a stage-gate release model. Business was driven by the development of products for mass customization.

Company B was a manufacturer of product lines of utility vehicles. In this environment, the teams were partially implementing ASD (Scrum). Some competences

were separated, e.g. System Engineers sat separately and deliver specification written on documents. Special customers requesting special features drove the business, and speed was important for the business goals of this company.

Company C was a manufacturer of telecommunication systems product lines. Their customers receive a platform and pay to unlock new features. The organization was split into cross-functional teams, most of which with feature development roles. Some of the teams had special supporting roles (technology, knowledge, architecture, ect.). Most of the teams used 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 before the feature was deployed.

Table 12. Studied cases and different context parameters.

	Case A	**Case B**	**Case C**
Product description	Several components of automotive embedded software.	Communication system, signal elaborating system, GUI system and customized OS platform.	Components and features development for a telecommunication platform.
Product lifecycle	6-7 years	> 15 years	~15 years
Business Model	International Markets	International Markets and individual customers	International Markets
Processes	• Scrum based implementation – Several weeks iterations • Input: requirements from different product owners/different projects • Output: deliver a *platform* to testers to be validated and verified	• Scrum based implementation – Several weeks iterations • Input: specification System Engineers • Output: delivers a *subsystem* to separate unit of testers to be validated and verified	• Scrum implementation – few weeks iterations • Input: white box specification from System Engineers • Output: deliver a component to be integrated by other units developing other products
Organization (unit of analysis only)	• 10-20 developers (including architects) • Cross-functional teams • Testers in a separated team • Requirement Engineers in separated business unit	• 2-25 developers (depending on the project, including architects) • Organized in functional teams • Testers in separated team • System Engineers in separate team	• 20 developers (including architects) • Cross-functional teams • System Engineers in separated team
Outsourcing	Some components outsourced	No outsourcing	No outsourcing

6.3.3 Data Collection

Our data collection followed three steps, in which we alternated several methodologies. First we employed a set of in-depth interviews for exploring the challenges in the achievement of goals based on speed. In the second step we validated and prioritized the interaction challenges through the collection of quantitative data using a survey. We also used the survey for collecting qualitative data and exploring which spanning activities were employed or missing in the companies. Finally, we collected validation responses through focus groups.

6.3.3.1 *Problem investigation: what challenges are decreasing speed?*

During the first step we conducted seven in-depth, semi-structured two-hours-long interviews with informants from each company. Semi-structured interviews are usually suited for exploratory studies, in which the researchers need to gain new insights on a novel phenomenon and strive to understand the factors involved. [62], [68].

Informants were chosen on the basis of their role and expertise, and the aim was to understand the issues by inquiring key employees rather than collecting a large amount of more superficial data. This allowed us to ask follow-up questions and therefore to follow multiple leads. All informants were senior engineers or managers, and had experience with the implementation of agile processes. For case A we interviewed an architect and a line manager from the same project. For case B, we interviewed four developers from four projects, some of whom had also architect or product line management roles. In case C, we interviewed one informant with the role of both senior developer and architect.

The interview protocol had fixed questions about the informant's role and context, and open-ended questions to bring out experiences related to challenges when reaching several kinds of speed.

Interviews followed the natural flow of discussion, and we included questions to cover predefined themes. Two main themes in the interview guide were represented by "agile practices" and "reuse practices". Both of them were meant to capture key practices clearly aimed at different business goals based on speed.

6.3.3.2 *Validation, prioritization of the challenges and missing practices*

The challenges revealed by the previous collection method, which relied only on qualitative information, needed to be strengthened by a more quantitative validation. Also, we wanted to understand which of the challenges were more severe in the companies and therefore needed to be prioritized. Finally, we wanted to understand what practices (spanning activities) were used or missing in different contexts.

For this purpose we created an on-line questionnaire that was designed using a combination of closed- and open-ended questions [69]. The first version of the survey was tested by two PhD students, to have feedback from an academic point of view and by at least one contact in each company, to make the questionnaire fit the jargon and being understandable for all the respondents. During the test we also monitored time and caught misunderstandings due to formulation of the questions.

The motivation for using an on-line questionnaire was to maximize coverage and participation without having to set up a time-consuming interview system. A survey is a suitable method for data collection when asking structured questions [68], [69]. The participants could access the survey whenever they wished. The answers were bound to a web-link and a mail address, so the respondents could also interrupt the survey and continue it in another moment. The questionnaire was made accessible on-line at surveymonkey.com between April and June of 2012.

The questionnaire was structured into two parts:

- two questions about the role of the respondents and their organizational unit of work. Different names, dependent on the context, were reported for the same role. We decided to abstract the wide set of roles into "manager", "system engineer", "designer" and "tester". We have chosen this categorization because of the frequent mention of such roles during the interviews and we could easily map these roles to clear references in software development literature.
- a set of 23 interaction challenges. For each one, the respondents were asked to give an answer among four alternatives: "No, I don't recognize the challenge", "Yes, I

recognize the challenge in other units", "Yes, I recognize the challenge in my own unit", "Yes I recognize the challenge in my unit and other units". When appropriate, we added a fifth answer, in which the respondents could express their inability to answer due to a lack of experience of the described situation. Such alternatives allowed us not only to validate the answers but also to obtain the perceived spread of the challenges in the organization. For each challenge we added an open-ended question using a text area. We asked which practices were applied or missing by the practitioners in order to handle the challenge.

Table 13. Challenges found in the first step and in-depth studied in the second step.

Q01	The processes/ways of working that you have to follow are not suited for the kind of product that you are developing
Q02	The processes/ways of working that you have to follow are not suited for the project management
Q03	Project-related defects (defects in the same reused component fixed in some project but not in others)
Q04	Developers and system engineers are not co-located, which causes communication problems in requirement agreement
Q05	There is an upcoming product. Erroneous assumptions have been made on what part of the existing software can be reused and/or adjusted, causing inaccurate budget or resources allocation
Q06	Evaluation of costs and feasibility of new features don't take into account implementations and constraints, causing inaccuracy in budget and resources (for example time or workload) allocation
Q07	There is no time to improve parts of software shared among projects.
Q08	A satellite unit is "invisible": for example, it's difficult to consult them, there is no clear information on their work or cannot be guided properly (the kind of problem may depend on your role)
Q09	A development unit has to build a component to be integrated in other units' (projects') system, but the communication with them is not sufficient.
Q10	A development unit was forced to integrate a common component. This caused communication problems, and now the unit is not willing to integrate new common components.
Q11	Reuse is not supported by the Product Line Management, it's an individual initiative.
Q12	Different attitudes and values of distributed (not co-located) teams (or units, projects) caused communication problems.
Q13	Team's (or unit, project) lack of will to integrate a common component.
Q14	Lack of understandable documentation or of proper communication causing long warming up periods for consultancy or for new employees to understand the system
Q15	Documentation is abundant but it doesn't help to understand the code
Q16	Leaders' mindset is not open to listen and they are not able to recognize strengths and weaknesses. This hinders the development of improvements.
Q17	Different favorite programming languages in the same team create communication .
Q18	Different favorite tools in the same team create communication problems.
Q19	Artifacts received from the system engineers are not clear enough because they were created with inappropriate tools
Q20	Developers are too constrained by system engineers on design (for example, developers receive white box/very detailed specification)
Q21	An internal interface had to be exposed to other units to allow their development. The interface documentation provided didn't help to understand the code.
Q22	Disagreement between different development units (or projects) about what set of functionalities a common component should provide. This creates communication problems
Q23	Loss of knowledge about a reused framework's variation points, for example a framework created some years.

The 23 challenges are the results from the previous exploratory interview study conducted in the same companies, in which such challenges were categorized as related to interaction. We operationalized the factors in a closed-question-survey. In some

cases we decided to split the factor into two questions. For example, the inhibitor "different tool and programming languages" was split into two questions, Q17 and Q18. For most of them, however, there is a one-to-one correspondence between the factors and the questions in the survey.

A 22-pages survey guide with detailed explanation for most of the factors, complemented with concrete examples, was made accessible for the practitioners through the web.

The challenges, which represent also the results collected in the first step, are outlined in Table 13.

6.3.3.3 Sample for the survey and response rate

We obtained 36 complete answers. The web tool helped us to select only the answers that came from respondents who finished the whole survey, i.e., incomplete answers have been discarded.

We had an overall response rate of 68% for the quantitative answers, i.e. 45 out of 66 total email invitations sent out, of which 36 were completed answers. This entails that 80% of the participants who started the survey finished it, which was, in the end, 54% of the total invited participants.

Figure 27 shows the distributions of answers across roles. We can see how all roles were covered: company B and C respondents cover all roles, whereas company C respondents cover all roles except testers. We notice a major presence of designers, especially for company C.

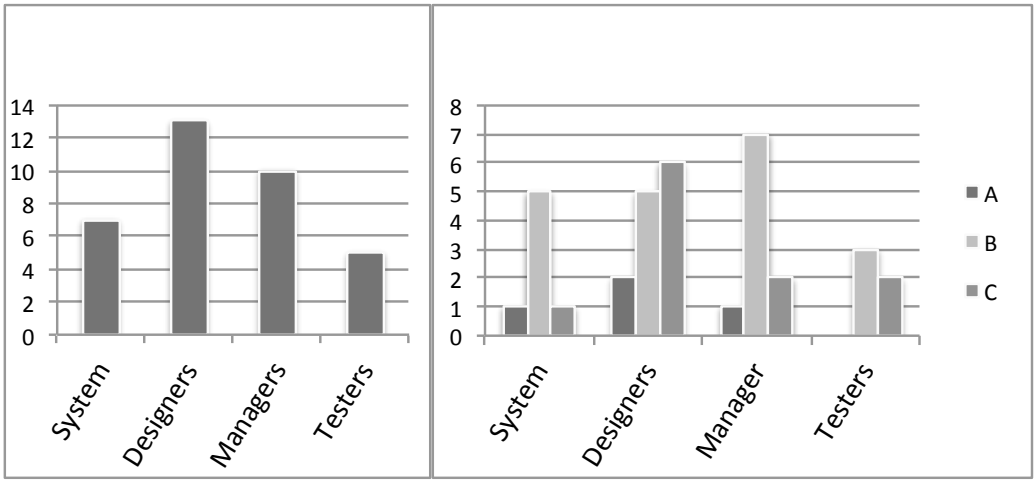


Figure 27. Sample of the respondents, divided by roles and companies.

6.3.3.4 Validation of spanning activities: collective interviews

In order to validate our results, we organized three collective interviews, one at each company. Most of the practices were recognized and we have excluded the ones rejected by the employees.

6.3.4 Data Analysis: combination of qualitative and quantitative analysis

We analyzed the data using a combination of qualitative and quantitative approaches (*methodological triangulation*), as recommended in [63]. This choice is also regarded as "strong analytic strategy" in [62].

6.3.4.1 *Grounded theory*

For the qualitative data, the interviews from the first step and the open-ended answers in the survey, we used an approach based on Grounded Theory [55], [61]. Instead of starting with a hypothesis and then testing it, it's based on a bottom-up process in which the theories (and the hypotheses) emerge from the data and therefore are grounded in it. The aim is to avoid the researchers to fit the data to preconceived hypotheses. The method used is also focused on validating theories by the systematic comparison across the data, the codes and the concepts.

We coded the text using a tool for qualitative analysis (Nvivo), which allowed us to maintain traceability between the data and the produced codes. The codes were categorized and sorted in order to link, compare and combine them into an organic representation of the investigated data, which led us to identify novel theories or to confirm existing knowledge.

6.3.4.2 *Coding*

We first analyzed the audio from the interviews and the text from the open-ended answers in the questionnaire, slicing it into small pieces and linking the relevant ones to quotations and codes. This task was carried out by the first researcher and checked by the second one, and was part of the *open coding,* in which we followed Strauss and Corbin's method [55]. In this phase, the first researcher summarized and interpreted the informants' words. Some factors were the result of in-vivo coding, where we quoted the interviewee; this was done in cases where the concept was well explained: *"Improvements depend on leaders mindset, open to listen and recognize strengths and weaknesses"*. For other factors we summarized from the source, "Long warming up periods for consultancy" summarized *"[…] is very hard for a consultant coming in […] we see that we have a half-year initial period just for [tool name] tool; it's another half year or year for the product to come into that to know it on a pretty good level […]"*. Yet other factors, are referring to many statements, which are showing the specified clear issue, only if considered together: "Business side afraid of upfront investment for a software product line".

6.3.4.3 *Categorization*

We then categorized the codes according to our research questions. In the first step, dedicated to problem identification, the categorization revealed the need for several hybrid categories for special influences, such as influencing several kinds of speed with different polarities: for instance, "Lack of detailed documentation" has a negative influence on *first deployment speed* but a positive influence on the *Replication Speed*. The categorization depended on the perspective that we took: for example, in the first step we gathered all the challenges concerning interactions under the "Interaction" area of improvement and its sub-categories. In the results we show how "interaction" factors (with negative influence on speed) were outnumbering the other factors, motivating the follow-up survey.

6.3.5 *Quantitative Analysis*

We performed statistical analysis on the answers to the close-ended questions in the questionnaire, using the tool SPSS. The purpose was to validate and rank the challenges, for which we used frequency analysis, and to correlate the answers to roles and companies by using Pearson chi-square tests. We could not perform more parametric tests since the size of the sample was not large enough to provide meaningful statistical evidences. The results are shown in Table 14.

6.3.5.1 Frequency Analysis

We have used frequency analysis for the *recognition* of a factor. We have grouped the factors in *strongly* recognized and *weakly* recognized, depending on the ratio of the participants answering "yes" or "no". However, when the number of yes and no was similar, we considered the factors as *controversial,* which means that even if the factors are strongly or weakly recognized, there was contrast in the answers. Combining these two categorizations, we had four categories: *strongly recognized*, *strongly but controversial*, *weakly but controversial* and *weakly recognized* factors.

We have also explored the *perceived spread* of a factor based on the respondents' experience of the issue being not only local (for example, present in the respondent's team) but affecting other parts of the organization close to the respondents. In order to evaluate the spread of the issues through the organization, we had to weight the four answers provided by the respondents. We obtained ordinal data similar to a Likert scale without a middle-point. We have weighted the answers with the following weights: zero was mapped to "no", one to "yes in other units/teams", three to "yes in my unit" and four to "yes in my unit and in other units" (1+3). We have weighted the answer *yes in my unit* as three times *yes in other units:* this way we "trusted" the direct experience more than the participants' perception about the presence of the factor in other parts of the same organization. We have emphasized the polarity between the "no" (zero) and the "yes" (3): in this way the means around three would show strong recognition, means around zero strong negation, and means around two would show high polarity in answers (not many neutral answers). The one point assigned to *in other units* weakly influences the means and is useful to give us an idea of how much the factor is perceived as spread throughout the organization. We could have assigned a higher weight to *in other units* and the results would have given a stronger result on the spread of the factors, but we have chosen to be cautious.

6.3.5.2 Chi-square tests

To discover if the answers on the factors depended on the company, we have used cross-tabulation, and performed a Pearson chi-squared test to check if the distribution of the answers changed with respect to the companies B and C (we couldn't use A because of the few respondents). For factors that gave a significant response (p-value < 0.1), we could find evidence that there were differences in the expected distribution. We performed the same test using the roles as categories. Finally, we combined the categories, roles and companies, in another cross-tabulation, but the single cells contained too few entries to perform a useful analysis. The selected and detailed data relevant for the explanation of the results are displayed in Table 15, Table 16, Table 17, Table 18 and Table 19.

CHALLENGE (abbr.)	ID	RECOGNITION		RANK WITH SPREAD		DEPENDS ON CONTEXT	DEPENDS ON ROLE
		%		MEAN			
Developers and system engineers are not co-located	Q04	93.9	Strong	3.52			
Processes/ways of working not suited for the kind of product	Q01	83.3		3.13			
"Invisible" satellite unit	Q08	84.2		3.11			
No time to improve parts of software shared among projects.	Q07	85.3		3.00		x	
Lack of documentation/communication, long warming-up	Q14	86.1		2.94			
Not sufficient communication for reuse	Q09	84.8		2.94			
Different attitudes and values of distributed teams	Q12	80.6		2.72			!
Erroneous assumptions on reusable software	Q05	77.8		2.67			x
Project-related defects	Q03	75.0		2.44		x	
Abundant documentation does not help understanding the code	Q15	64.7	Strong, Controversial	2.38			
Reuse individual initiative	Q11	61.1		2.14			
System architecture artifacts not clear enough	Q19	55.6		2.08		x	
Disagreement on features in reused software	Q22	60.0		2.03			
Loss of knowledge about a reused framework's variation points	Q23	54.5		1.91		x	
Different favorite tools in the same team	Q18	54.3		1.89			
Estimation of new features without implementations constraints	Q06	52.8		1.86			!
Processes/ways of working not suited for the project	Q02	54.3		1.86			
Leaders' mindset is not open to listen	Q16	50.0		1.64			
Developers are too constrained by system engineers on design	Q20	52.8		1.64			
An internal interface had to be exposed to other units	Q21	50.0		1.53			
Different favorite programming languages in the same team	Q17	36.7	Weak, Contr.	1.33			
Team's lack of will to integrate a common component	Q13	38.2		1.26			!
Development unit was forced to integrate a common component	Q10	29.0		.84			x
⊘	⊘	0.0	Weak	0.00			

Table 14. List of challenges ranked by recognition, spread and their dependence on context or on role.

Table 15. Number of answers for Q03 by company, followed by the Chi-Square result.

Q03- Project-related defects				
Answer Company	No	Yes, my other units	Yes, my unit	Yes my and other units
C	6	1	3	1
B	2	1	7	11
Pearson Chi-Square				0.021

Table 16. Number of answers for Q07 by company, followed by the Chi-Square result.

Q07- There is no time to improve parts of software shared among projects				
Answer Company	No	Yes, my other units	Yes, my unit	Yes my and other units
C	4	1	1	4
B	0	0	8	13
Pearson Chi-Square				0.005

Table 17. Number of answers for Q23 by company, followed by the Chi-Square result.

Q23- Loss of knowledge about a reused framework's variation points				
Answer Company	No	Yes, my other units	Yes, my unit	Yes my and other units
C	4	0	2	0
B	5	1	1	7
Pearson Chi-Square				0.1

Table 18. Number of answers for Q05 by role, followed by the Chi-Square result.

Q05- Erroneous assumptions on reusable software				
Answer Role	No	Yes, my other units	Yes, my unit	Yes my and other units
System eng.	1	0	1	5
Designer	4	0	5	4
Manager	0	2	3	5
Tester	3	0	1	1
Pearson Chi-Square				0.1

Table 19. Number of answers for Q10 by role, followed by the Chi-Square result.

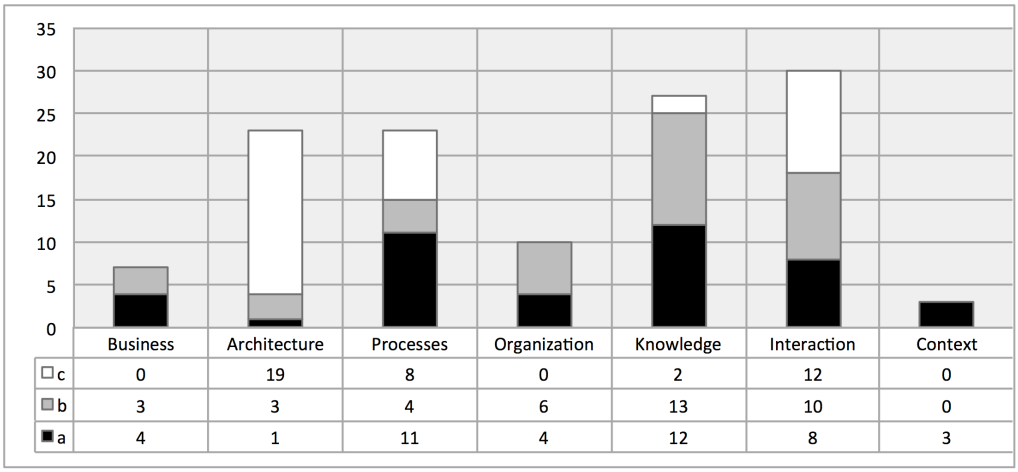
Q10- There is no time to improve parts of software shared among projects				
Answer Role	No	Yes, my other units	Yes, my unit	Yes my and other units
System eng.	4	0	2	0
Designer	11	0	1	0
Manager	2	2	1	2
Tester	5	0	0	0
Pearson Chi-Square				0.027

6.4 RESULTS AND ANALYSIS

6.4.1 *Interaction challenges hinder the achievement of business goals based on speed in all studied contexts*

With the first step we found the challenges hindering the achievement of business goals based on speed. The qualitative investigation brought to light 114 factors [71]. After the categorization of the factors, we obtained a distribution of the factors in the areas of improvement showed in the bar charts in Figure 28. The diagrams display which areas the informants were most concerned with, within a single company and cross-company.

Figure 28. Bar chart of all challenges. Interaction is the most populated category and also contains an even distribution of the challenges.



	Business	Architecture	Processes	Organization	Knowledge	Interaction	Context
c	0	19	8	0	2	12	0
b	3	3	4	6	13	10	0
a	4	1	11	4	12	8	3

The results show that interaction challenges had an impact on first development speed, replication speed and evolution speed:

- *First Deployment Speed:* when a set of features is released for the first time, the speed is affected by the interaction speed among the teams that have to integrate the features. This kind of speed has an impact on *hitting the market fast* to anticipate the competitors. Fast deployment speed also *shortens the loop in market testing*.
- *Replication Speed:* when a feature is embedded in a previous release, interactions are needed between the team responsible for the new features and the teams that had developed the former ones. Replication increases ROI when the effort made for the first deployment speed is spread on the *release of new products and services based on the existing software.*
- *Evolution Speed:* when a feature needs to evolve after its release, the changes will affect other features, which requires interactions. The speed in *reacting upon a change request* can be critical for gaining the trust of the customers.

The even distribution of the 30 factors in the *interaction* column of Figure 28, (12, 10, eight for case A, B, and C), emphasizes that interactions is a common concern at all sites.

These results called for further investigation of intra-group interaction challenges. The category containing such factors was the most populated one, which showed its importance for the respondents. This was also the category with the most evenly distributed factors among the studied sites, which suggested a high degree of generalizability of such factors.

6.4.2 *Validation and prioritization of interaction challenges*

In the second step we conducted a survey to validate and prioritize the discovered interaction factors.

Through a quantitative analysis of the answers, we produced an ordered table with the factors, their recognition in terms of frequencies and their perceived spread in the companies. The analysis of the data also highlighted factors recognized by the respondents in some contexts but not in others, and factors recognized by some roles but not by others. The results are summarized in Table 14.

The second column from the left (*Recognition*) displays the percentage of respondents (calculated on the valid answers provided) that have recognized that factor. All the factors are recognized at least by 29% of the participants. In the middle column we highlighted the level of recognition (*strongly, strongly but controversial, weakly but controversial and weakly recognized*) with bold borders grouping the rows of the factors included in each cluster. Nine factors, with more than 75%, are strongly recognized. The factors with more than 50%, another 11, are strongly recognized but controversial, which means that there are more "yes" than "no", but we found some disagreement in the answers. The last three factors are weakly recognized but controversial, which means that the number of "no" is larger than the number of "yes", but some of the respondents considered the challenge present in their context. None of the factors were considered weakly recognized.

In the middle column (*Rank with Spread*) we show the spread calculated using the means of the answers weighted from zero to four: such results, together with the recognition, illustrate how much the respondents perceive the spread of the factor in other parts of the organizations close to them. This partially changes the order of the factors, as we can see for Q14 and Q01: the former one has a higher direct recognition rate, whilst the latter one is acknowledged by less respondents than Q14 but is been considered more widespread within the company. Notice that the spread doesn't influence the previous categorization of the factors (*strongly*, *weakly*, etc.).

In the last two columns from the left (*Depends on Context* and *Depends on Role*) we have marked the factors with a "x" when we have statistical evidences that the factor has the property.

6.4.2.1 *Context-related challenges*

The distribution of the answers with respect to the respondents' company didn't change for 20 of the factors. This means that we could reject the hypothesis (as explained in the section 6.3.5.2, the significance was 0.1) that the recognition of most of the factors depended on the company of the respondents only for three of the factors (Q03, Q07 and Q23). The results are shown respectively in Table 15, Table 16, and Table 17. These challenges are reported in Table 14 with "x" on the column *Depends on Context*. In the followings we describe the differences among the companies.

Q03: *Project-related bugs or defects*

For factor Q03 there is a great difference between company B and C: the respondents in company C don't recognize the factor, while the ones in company B strongly recognize it.

Q07: *There is no time to improve parts of software shared among projects*

For factor Q07 the respondents in company B strongly recognize it, while the other ones stand in the middle. This shows that this factor is strongly present in company B while is controversial in the other companies. With respect to Table 14, this partially weakens the strong recognition of Q07 since the mean is influenced by company B.

Q23 - *Loss of knowledge about a reused framework's variation points, for example a framework created some years before.*

In company B more than half of the informants (eight) strongly recognize this factor. Even though the result is "strong but controversial", it differs from company C, where only two respondents recognized the factors.

6.4.2.2 Different roles' view on some challenges

We found that the answers changed for only two factors (Q05 and Q10) when given by different respondents with different roles. The results are shown respectively in Table 18 and

Table 19. These challenges are marked with "x" in Table 14 in the column *Depends on Roles*. The mark "!" signals the presence of evidences in the data that need further validation.

Q05 - *There is an upcoming product. Erroneous assumptions have been made on what part of the existing software can be reused and/or adjusted, causing inaccurate budget or resources allocation (for example time or workload)*

For this factor, managers and system engineers expressed a clear propensity to recognize the issue, while designers and testers gave more controversial answers.

Q10 - *A development unit was forced to integrate a common component (shared with other units). This caused communication problems and now the unit is not willing to integrate new common components.*

6.4.3 Boundary spanning

In the previous sections we have identified and ranked the interaction challenges. In Figure 29 we show the critical boundaries in need of interaction improvement. As explained in the methodology section, we list the spanning activities related to the mitigation of interaction challenges at each boundary. Each activity is defined according to a specific boundary, a frequency (*project, iteration* and *ad-hoc*) and involves a coordinator and a spanning object.

In the following section we perform an in-depth analysis of the critical boundaries and the spanning activities that are currently needed for avoiding the interaction challenges.

6.4.4 Prioritization of boundaries for spanning activities

The rank of challenges combined with the qualitative analysis of the textual answers show a number of critical boundaries in need of spanning activities, between the agile team and other social groups:

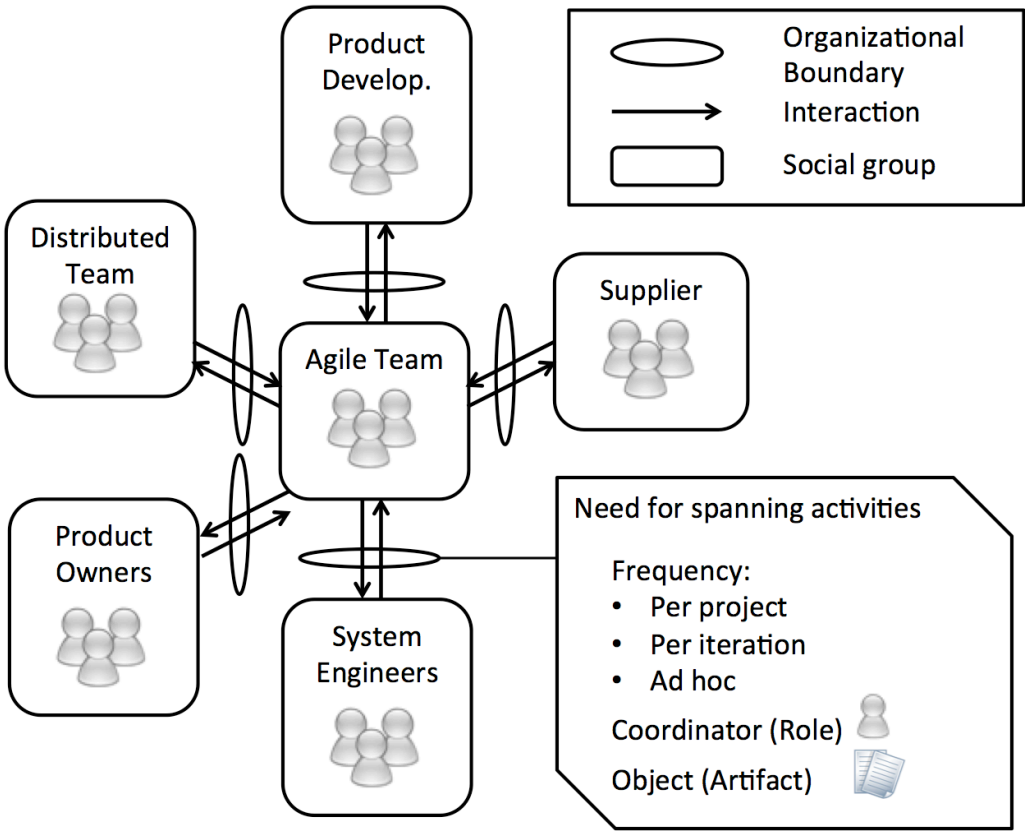


Figure 29. Critical organizational boundaries in need of spanning activities. Each spanning activity has a frequency, a coordinator and it might involve an object.

Team - System engineers. Q04 is the most recognized factor, almost by all the respondents (93%, which include many system engineers): system requirements and architecture should be continuously discussed between system engineers and the agile team. This is especially important for software reuse, which has to be recognized at a system level and mapped to the agile organization in an optimal way.

Team - Product development and management: the respondents' answers about factor Q01 and Q02 (process-related communication) are quite different. It seems that ASD suffers particularly when the agile team needs to interact with the product development rather than with the project. This aspect might be connected with the kind of developed product (i.e. embedded systems): hardware's design and development process follow a more waterfall-oriented model, which conflicts with the iterative one claimed by ASD. In general, software development is bound to the overall product development process, which has longer iterations than the ones prescribed by ASD and therefore hinders the responsiveness that would be gained by employing ASD at a team level. However, we cannot say if this is a real constraint or a factor related to the legacy of the product development in place. In the latter case, we hypothesize (supported by the qualitative answers) that a change in the product development process to make it more ASD-friendly (for example by providing frequent interfaces for strategic input) would foster the interaction speed of teams with the product management.

Team - Distributed teams (Q08): the speed of the agile team might be hindered by dependencies to other distributed teams. This includes suppliers commissioned to develop outsourced software components but also teams distributed in several buildings. Especially, when the quantity and the channels of communication are constrained by contract terms such as time intervals or approval requirement, speed is hindered by the occurrence of strong delays. The same, but less drastic issues, occur by mismatches in processes, practices, attitudes and values. Frequent meetings on-site and social network activities would ease this interface.

Team - Other projects' teams: Q07, Q03 and Q09 are strongly recognized, but the results have been stronger for a specific context (Q07, Q03 for case B). The isolation of the projects, especially in terms of budget and resources, create silos that hinder reuse across the projects. The exclusive focus on one project by the team leads also to hindering the communication (and speed) among teams in parallel projects: this creates inefficiencies, especially if there are dependencies among the projects, and strategic reuse always brings such dependencies. Solutions such as relocation in other projects have been proposed in the qualitative answers, to spread knowledge of the system and to gain acquaintance among the employees.

Team - Product Owners: factor Q05 shows that software reuse can be erroneously considered during the selling process (or the marketing scope). In the qualitative answers, respondents specified that no resource allocation was estimated when reusing *similar* components. However, the business value of identical components seems to be much higher than that of the similar ones. This suggests that full reusability of components should be checked by the Product Owners. This would encompass communication with the involved teams responsible for the components. The strong recognition of this factor mainly by management and systemization shows that designers and testers are not fully aware of strategic and business related goals. This suggests that if the agile team is not guided continuously by strategic inputs, it might incur in sub-optimal decisions.

6.4.5 Boundary spanning for each project

Once we identified the challenges and the critical boundaries, we analyzed the qualitative answers of the participants in order to find spanning activities that were used or missing for the whole project.

Several informants mentioned the need for activities at the beginning of the projects.

"*Everyone start off being very busy with their own stuff and don't have time to talk to others. Eventually this leads to a crisis in the project where the schedule slips and integrations fails.*"

The most common means to achieve synchronization is the organization of workshops. The data suggest that the workshops should be focused especially on the following actors and activities.

Architecture between the *team* and the *system engineers (architects)*: architecture is a synchronization mechanism and is important for achieving platform and component reuse among several projects in all investigated cases. The architecture is also seen as a *boundary object*, and the production of such object needs to be agreed upon. The current implementation of ASD seems to suffer from the lack of suitable spanning objects, for example, the architecture documentation. This documentation is currently not fulfilling the needs of the stakeholders.

"*Better coordination and planning of how architectural changes should be introduced in a product with high feature growth. The changes should be introduced in steps and following a plan agreed to by all concerned parts"*

A workshop in the beginning of the project would help agreement on the stakeholders' "requirements" for the necessary boundary object (both the team and the architects).

"*Some defined level of documentation describing the system/subsystem for a newbie, requirement engineer to more easily get into/understand the system*", but "*have less complimentary documentation, in best case only some pictures to explain the overall idea*".

Furthermore, the discussion would increase the team's awareness about the product and product line (through the architects' view) and ease the follow-up communication with the architects with the increase of personal acquaintance. No coordinator role has been identified, but we hypothesize that the architects would be the coordinators for this activity.

Expectation between *distributed development teams* and between the team and the *suppliers*: according to the informants, a clear challenge is, for a team, to know what to expect from other teams. Teams need also to agree on future spanning activities in order to avoid hindering each other during development and to find suitable points for synchronization in the customized processes.

Negotiation and awareness *between teams, architects and product owners*: what the respondents lament is the decision making based on wrong assumptions in the scope analysis which would be reflected on the pressure suffered by the team to meet deadlines. Activity tools such as planning poker is used *within* the team, for example for estimating the cost of feature development, but we found the need for a workshop covering the feasibility and the expectations (negotiation) on higher level development capability, such as architecture goals (including other projects) and time-related penalties to be agreed with the customers. This kind of workshop would also increase information about business strategies in the development team, making them closer to the customer perspective.

6.4.6 Boundary spanning for each iteration

Spanning activities aligned with iteration frequency can be used as a synchronization mechanism among several teams and between the team and other kinds of organizational groups [22]. However, often the respondents mention the lack of coordination among interacting teams and between the team and other groups.

Process mismatch among teams: A major challenge in the former case is the lack of organized opportunities for the teams to meet among themselves. Such problem is also connected by the respondents to the mismatch of processes: the processes are usually customized by the teams, for example the development of two features might start in different times and the iterations and development phases within the iteration might not be synchronized. Such challenge has been especially emphasized when an agile team had to be in contact with external suppliers not complying with ASD (case A and C).

***Process mismatch between the team and other groups*:** As for the coordination between the team and other groups, a main challenge is the mismatch of processes between the agile team, organized in iterations, and the rest of the organizational groups/other organizations, which is usually not. The product development for all the cases is bound to the creation of the actual physical system on which the software has to be deployed. Consequently, the agile team lacks interaction with other engineering experts (e.g. system engineers), product management and other resources that follow a different process and therefore are not available. As an example, it might happen that a feature is not tested as soon as it's been developed.

"*A feature should be verified as soon as it has been implemented (and not a few months later) to get fast feedback, save time and prevent a lot of bugs at the end of the project. - When creating parallel implementation tracks in the same product/project there should be a plan for how the tracks should be merged together.*"

Spanning objects: generic process guidelines to be reused.

Coordination role: The spanning activities would include the presence of a coordinator in the other groups, that would facilitate the communication with the agile team by being available at the end of each sprint with the necessary information. For

example, a system engineer or an electrical engineer should be appointed for the interaction with the agile team.

***Coordination among the teams*:** Many respondents argue that there is a need for making the team responsible for their interaction with other teams. Such interaction is usually due to socio-technical dependencies. In some cases such interaction might be regulated through social links developed among the teams, through communities of practices [105] or might be initiated by a Scrum Master (e.g. using Scrum of Scrums [92]) by organizing inter-team meetings each iteration. However, the informants usually consider part of the documentation necessary for coordination, since only face-to-face interactions seem not to be enough in a large environment.

"*Insert documentation stories into the backlog. Include documentation into definition of done.*"

Spanning objects: The activity is to include the creation of documentation as a task/story in the backlog. Such boundary activity would assure the prioritization of the boundary object (documentation) creation through the use of another boundary object, the backlog itself.

Coordinator role: in the answers related to process mismatch, there is usually the description of a needed activity but not the role responsible for it.

6.4.7 Ad hoc boundary spanning

Shared responsibility of integration: the respondents highlighted the need for having clear responsibilities for the integrated parts of the software. Currently there is a lack of clear reference about who should be responsible for implementing bug fixes, for improving the code (including architecture) and for explaining past decisions. The integrated "whole" might be a component (in case features spread to several components) or a set of connected functionalities.

"*Take responsibility for "your" product", "[…] very clear who is doing what and when everybody is supposed to deliver what and to whom. The overall objective of each delivery and the limitations (not implemented functionality) should also be communicated to all involved parties*"

"*Promote the attitude of taking responsibility for the WHOLE chain of functionality*"

Respondents suggest a spanning activity consisting of monitoring and recognizing such situations in which responsibility could "fall between the cracks", and reacting by defining responsibilities through a meeting with the involved actors.

"Review interfaces between different processes e.g. System design - SW Design to enable a more continuous flow of information"

However, also in this case, a coordinator has not been identified by the respondents. Such an activity could probably be carried out at each iteration, but according to the respondents, it's enough to monitor and react in an ad hoc way.

6.5 DISCUSSION

Our research focused on 23 challenges related to achieving business goals dependent on speed. The emphasis is on interactions across organizational boundaries, specifically between the agile team and other parts of the organization. Our findings are focused on large organizations developing embedded software and employing ASD.

The results show several and widespread interaction challenges influencing business goals dependent on speed. Highlighting the most recognized and spread interaction challenges would help practitioners in selecting the targets for investing resources and would provide guidelines for further research. Showing the relationship between

challenges and context warns practitioners about known issues in applying ASD in large companies developing embedded software. Showing the different views among the roles on some factors reveal possible existing conflicts or lack of awareness about some of the challenges, for example the lack, in the teams, of strategic goals that need to be communicated by product managers and architects. Highlighting which practices are implemented or missing would help organizations in defining a coordination strategy that would include spanning activities, objects and coordinators in order to mitigate interaction challenges and therefore eliminating delays in the reaching the business goals.

In the following sections we discuss the generalizability and contextualization of challenges and activities. Then, we compare the results to the cases found in the literature [22] in order to highlight our contribution with respect to related work and to answer RQ4. Finally, we discuss the main limitations and threats to validity and present more related work.

6.5.1 Generalization and contextualization of challenges

The results show an overall recognition and spread of the challenges. 20 challenges have been strongly recognized, of which 11 were still controversial. Only three challenges were weakly recognized but controversial. No challenges were recognized as weak without any controversy. These results allow us to make a further step towards the generalization of the found challenges. In fact, there is convincing evidence that companies B and C equally agreed on the recognition of the challenges. With less evidence, we can consider company A on the same line. This means that Table 14 is equally informative for the companies.

As for Q03 Q07 and Q23, the recognition varies according to the context, and Table 14 shows this together with the ranking of the challenges. These factors are strongly related to context B. In this company projects were isolated, hindering the employment of an overall strategy for reuse. The distinctive parameter related to context B is related to the business model: individual customers with different needs and different time constraints may strongly hinder the investment needed for the creation of a software product line. This is shown also by the recognition of Q23 mainly by case B: according to the qualitative data, the business value of the ad-hoc reuse was overestimated. If the component was not reused exactly as originally implemented, even slight changes would bring a lot of work to re-understand the implementation and to test the small changes that could affect the behavior.

6.5.2 Generalization and contextualization of activities

The consistency among the respondents' answers for these three cases is quite high. We have not found controversial statements among the respondents, and the codes and categories used for this analysis are linked to citation by at least two respondents. The only misalignment in the statements was related to the balance between the need of increasing boundary-spanning activities and team isolation. The actual amount of time spent in interaction is not known and the spanning activities need to be limited in order to allow the team to focus. Such topic, in our opinion, requires further research. The beginning of the project needs more time dedicated to spanning activities (e.g. communication of strategic goals to the team) and the creation of coordination objects (e.g. documentation shared by the teams), Even though it seems to be against the avoidance of a big upfront investment, as suggested by the ASD principles, other experiences reported in literature [117] refer to a "Sprint Zero" in which the team needs to understand strategic goals. The iterations are good opportunities for coordination and for instantiating or carrying out spanning activities.

Our results are not limited to one case only, but evidences from all three cases support most of the spanning activities reported in the results section. Such activities would help the creation, both for researchers and practitioners, of frameworks dedicated to embedded software development for complementing existing ASD practices. Especially the following areas are in need of improvements: the starting phase of the project, the development and maintainability of software architecture artifacts, the exchange of information between the team and the product owners and the interaction with distributed teams.

A relevant practice especially related to company B would be the creation of inter-project spanning activity for helping the agile teams, involved in several projects, in communicating similar strategic goals concerning software reuse. This activity and similar activities remain a challenge to be further studied, since the respondents could not propose a suitable solution. What we can report from our results is that if the business situation of the company includes non-synchronized customers, this kind of spanning activity should be an ad hoc activity. Coordinator and roles need also to be further investigated.

A major challenge in the current organizations is to find suitable roles: groups differing from the agile team appear to have different views and mindsets, which do not necessarily comply with ASD. This is hindering the development of boundary spanning activities and objects.

6.5.3 Comparison between embedded software development and pure software development

In order to show the contribution with respect to RQ4, we highlight the differences between our domain (large product companies developing embedded systems) and the one described in [22]. The paper is focused on coordination strategies in agile software development, which partially covers spanning activities as well. This paper is also the only one found by the literature review related to spanning activities. However, the companies analyzed in [22] are not large companies and they don't develop embedded software. We show which areas of improvement with respect to which contextual factors are different in our studied domain and we claim that we need further research dedicated to study new boundary spanning activities for such combination of context and areas.

We recognize four major contextual factors that are different between our cases and the ones described in literature:

- *Reuse*: the presence of heavy reuse (platforms, components) across the projects versus stand-alone projects.
- *Product development*: an overall product-development process is present in our domain but not mentioned in the cases in [22].
- *Customer*: in our domain the same product(s) is developed for multiple customers, who, in most cases, are not available (for example, the product is developed for a generic market). This is different from the cases studied in [22] where the organizations developed one product for a specific customer. In particular, in our cases we can differentiate between synchronized (cases A, C) and non-synchronized customers (case B).
- *Other disciplines*: in our domain other disciplines need to be connected to software development, such as electrical engineering, system engineering, etc. which brings in more stakeholders for the team than in [22].

The major differences in the results are the needs for more boundary-spanning activities, objects and coordinators in our cases, especially in the following areas:

- *Software architecture*: spanning activities, objects and coordinators for this purpose are strongly needed by the respondents, while they are not mentioned at all in [22]. We can relate this difference directly to the reuse factor, since software architecture is the main coordination mechanism for reuse.
- *Processes*: in [22] it is mentioned that the iterations might be used as synchronization mechanism. However, when a number of different processes *mismatch* but need to *interact*, there is a need for more spanning activities and coordinators. We can relate this issue with the context factors "product development" and "other disciplines", which we found are the main causes for having processes that mismatch.
- *Shared responsibilities*: in our results there are many references to the need of spanning activities for managing shared responsibilities across teams, while they are only briefly mentioned in [22], without in-depth research focus. Such responsibilities are mainly linked to the area of software architecture and therefore to the reuse factor (and therefore the replication speed goal), since the responsibilities are shared for integration and for the common understanding of the system.
- *Expectations*: spanning activities are mentioned in [22] for connecting with the customer that is not on site. In our cases, not only the customers are not on site, but there are also many at the same time for a given product (linked to the same contextual factor). This increases the need for boundary spanning activities and roles, in order to assure the convergence in understanding requirements among the teams.

Table 20 summarizes the need for new boundary spanning activities in embedded software companies employing ASD, which would avoid interaction challenges and would, therefore, improve speed.

Table 20. Comparison between embedded and pure software development according to current literature. The black cells with "x" show that we need novel spanning activities in the areas on the left column due to specific contextual factors.

Contextual Factor / **Area**	Reuse	Product Dev.	Customer	Other disciplines
Architecture	x			
Processes		x		x
Shared responsibilities	x			
Expectations		x	x	

6.5.4 Limitations and Threats to validity

This study has limitations: the ordinal data gained by the close- ended questions is based on the employees' perception about the presence of the issues in their unit or in other units, which cannot be considered a precise measurement. However, we have used appropriate statistical tools such as cross-tabulation and Chi-square analysis of distribution. Randomization of the sample cannot be completely assured, since the respondents have been conveniently selected by the companies themselves. However, we have explicitly recommended our contacts in the companies not to influence their choice.

The qualitative information is based on employees' statements and may be biased. However, the statements don't usually conflict with each other. On the contrary, the codes are often supported by quotations from several respondents and from several cases. Moreover, we have mainly discussed and drawn conclusions on factors that were strongly recognized, minimizing the possibility errors. The hypothesis and suggestions

for improvements have been selected among the most cited improvements coded during qualitative analysis.

We consider the threats to validity as presented in [63] and [118]. Possible threats to construct validity are: evaluation apprehension, mono-operation bias and background influence. We have handled these with the anonymity of the respondents, collecting data from three companies and we made sure that the respondents represented several roles, from managers to programmers. We limited the threats to conclusion validity such as instrumental flaws and influence posed on the subjects by running pre-tests with colleagues from academia and contacts in industries. As for internal validity, each respondent has taken the survey just once, while boredom was avoided by the possibility of interruption and continuation in a later moment. As for external validity (generalizability), the chosen cases allow us to generalize some of the factors and activities with respect to the commonalities in their contexts. By studying three cases we have partially limited external validity.

6.5.5 Other Related Work

Lindvall et al. [16] collected experiences from large companies (similar setting) in order to study the integration of ASD with standard processes already in place. The focus was in tailoring XP to suit the standard process and quality management. Under the category of "Cross-team communication support" the authors mention the presence of challenges in inter-team communication as an open issue. We have found more challenges and studied them in depth.

Karlström and Runeson studied the application of XP in stage-gate project management [101]. They conclude that such combination is feasible, but the agile team should interface with the gates, while at the same time the project management should adapt to support informal communication and documentation with the XP team. This is in line with our results: agile teams need spanning activities with product management and system responsible. The study [101] adds the suggestion, for the agile team, to adapt to the stage-gate model to coordinate with other teams.

Kettunen and Laanti [90] studied the large-scale organizational agility and proposed a framework for a company to embed an agile software project in a successful way. In the paper, the authors stress the need for an external process adaptable to the agile team, which confirmed our findings. However, the paper is not focused on other interactions that we have studied.

The paper by Lee [119] is an experience report on transitioning Large-scale project into agile. However, the paper is mainly focused on the intra-team perspective. As for the inter-team interactions, the authors propose an agile method for mail management and discuss the competition occurring among teams as a result of the introduction of metrics for assessing ASD.

On the basis of a literature review on ASD, Turk, France and Rumpe give their perception about the limitations of ASD [89]. They mention *Limited support for building reusable artifacts,* describing the importance of reuse practices, which bring especially long-term benefits. However, the authors don't investigate the issue in depth and don't provide any guideline. Our work focuses on this problem and analyzing the inter-group interaction factors related to it.

In [94] the focus is similar to ours, but the study has been carried out in one organization with several context, performing qualitative data analysis, while we have surveyed the challenges also in a more quantitative way in order to have a prioritized list of the most important ones. Moreover, we have studied specific interaction challenges related to several kinds of speed, which is not included in the scope of [94].

In [120] the authors present a set of interaction practices for Global Software Development. However, the study is not related to ASD. Some practices may be used to ease the factors studied in our paper, but such practices need to be adapted to an ASD environment.

6.6 CONCLUSION

In this paper we have studied inter-group interaction challenges influencing business goals dependent on speed. In particular, we have been focusing on large companies developing embedded software and employing Agile Software Development. We have thoroughly investigated the problem with a two-year case-study involving three large companies and several employees with different roles. We have employed both qualitative and quantitative research methods, with the aim of supporting the claims with a high degree of evidence triangulation.

We have studied 23 interaction challenges (Figure 28), most of which have been strongly recognized by the practitioners and they have been regarded as widespread in the companies (Table 14). Our results help practitioners in selecting targets for investing resources and provide guidelines for further research (RQ1, RQ2).

We have shown the relationship between challenges and different contexts, which warns practitioners about known issues in applying ASD in specific contexts, such as embedded software development and companies developing product lines (Table 20). The main result is the recognition of the need for implementing more boundary spanning activities for specific areas (RQ4). Especially in cases like company B, where reuse is an imperative business goal (replication speed) and has to be combined with the presence of multiple, non-synchronized customers, spanning activities need to be employed to mitigate the challenges between projects. Also, we have showed the different views among the roles on some factors, which reveal possible existing conflicts or lack of awareness about some issues.

Although we have explicitly asked the practitioners to share the best practices in use, we could see how many interaction challenges still miss a suitable solution, especially across specific critical boundaries (Figure 29). We build our results on top of existing research in coordination theory, and we provide guidelines by highlighting which organizational boundaries are needed for improving critical development areas (Table 20) and with which frequency (e.g. each iteration, project etc.). The results underline the importance of further research for defining new spanning activities, objects and coordinators to mitigate interaction challenges hindering speed in embedded software development combined with ASD (RQ3, RQ4).

Our future research is aimed at developing a framework including activities, roles and objects that would fill the gap recognized in this study. We are currently investigating the development of such framework in collaboration with several large companies.

7 ARCHITECTURAL TECHNICAL DEBT MANAGEMENT: TRADE-OFFS FOR AMBIDEXTERITY

Architecture Technical Debt is regarded as sub-optimal architectural solutions taken to improve short-term responsiveness but that would hinder long-term responsiveness. This chapter aims at improving software management by shedding light on the current factors responsible for the accumulation of Architectural Technical Debt and to understand how it evolves over time. We conducted an exploratory multiple-case embedded case study in 7 sites at 5 large companies. We evaluated the results with additional cross-company interviews and an in-depth, company-specific case study in which we initially evaluate factors and models.

We compiled a taxonomy of the factors and their influence in the accumulation of Architectural Technical Debt, and we provide two qualitative models of how the debt is accumulated and refactored over time in the studied companies. We also list a set of exploratory propositions on possible refactoring strategies that can be useful as insights for practitioners and as hypotheses for further research. We conclude how several factors cause constant and unavoidable accumulation of Architecture Technical Debt, which leads to development crises. Refactorings are often overlooked in prioritization and they are often triggered by development crises, in a reactive fashion. Some of the factors are manageable, while others are external to the companies. ATD needs to be made visible, in order to postpone the crises according to the strategic goals of the companies. There is a need for practices and automated tools to proactively manage ATD.

This chapter has been accepted for publication as:

Martini A., Bosch J., and Chaudron M. *"Investigating Architectural Technical Debt Accumulation and Refactoring over Time: a Multiple-Case Study"* Information and Software Technology, in press [121].

7.1 INTRODUCTION

Large software industries strive to make their development processes fast and more responsive, minimizing the time between the identification of a customer need and the delivery of a solution. The trend in the last decade has been the employment of Agile Software Development (ASD) [6]. 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 [38]. Tom et al. [33] 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) [31]. More precisely, ATD is regarded as implemented solutions that are sub-optimal with respect to the quality attributes (internal or external) defined in the desired architecture intended to meet the companies' business goals.

ATD has been recognized as part of TD in a recent (2015) systematic mapping study on TD [31]. However, such study highlights several deficiencies in the current body of knowledge: lack of reliable industrial studies, lack of focus on architecture anti-patterns and lack of studies involving the whole TD management process. In this paper we aim at filling such current gaps by investigating, in several companies, the overall

phenomenon of accumulation and refactoring of ATD. The study of such subject would also contribute to ASD frameworks, by highlighting activities for enhancing agility in the task of developing and maintaining software architecture in large projects [5].

In the context of large-scale ASD, our research questions are:

RQ1: What factors cause the accumulation of ATD?

RQ2: How is ATD accumulated and refactored over time?

RQ3: What possible refactoring strategies can be employed for managing ATD?

In this paper we have employed a 18 months long, multiple-case study involving 7 different sites in 5 large Scandinavian companies in order to shed light on the phenomenon of accumulation and refactoring of ATD. We have analyzed the qualitative data obtained from more than 50 hours of formal interviews complemented with continuous informal meetings with key roles involved in the architectural work, using a combination of inductive and deductive approach proper of Grounded Theory. We have qualitatively developed and evaluated a taxonomy of the factors to inform RQ1 and a set of models to inform RQ2. We have also derived some preliminary conclusions on which refactoring strategies can be applied and what effects they have to inform RQ3.

The main contributions of the papers are:

- A taxonomy of the causes for ATD: we present the factors for the explanation of the phenomena such as accumulation and refactoring of ATD. These factors might be studied and treated separately, and offer a better understanding of the overall phenomenon.
- Two qualitative models of the trends in accumulation and refactoring of ATD over time.
  - Crisis model – Shows the strictly increasing trend of ATD accumulation and how it eventually reaches a crisis point. We describe the evidences related to the occurrence of the crisis point and we connect such phenomenon to the different factors and their influence on the accumulation.
  - Phases model – Shows when ATD is currently accumulated and refactored during different software development phases. It 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 refactoring of ATD. Such practices would be aimed at delaying the crisis point.
- Possible refactoring strategies: we analyze how different refactoring strategies might lead to best- and worst-case scenarios with respect to crisis points.
- A detailed description of an additional and in-depth industrial case, which contributes to empirically evaluate the factors and to analyze the relationships among them in a specific context.

7.2 Architecture and Technical Debt

7.2.1 Definition of ATD

ATD is regarded [33] as "sub-optimal solutions" with respect to an optimal architecture for supporting the business goals of the organization. Specifically, we refer to the architecture identified by the software and system architects as the optimal trade-off when considering the concerns collected from the different stakeholders. In the rest of the paper, we call the sub-optimal solutions *inconsistencies* between the implementation and the architecture, or *violations*, when the optimal architecture is

precisely expressed by rules (for example for dependencies among specific components). However, it's important to notice that (in our studied cases) such optimal trade-off might change over time, as explained in this paper, due to business evolution and to information collected from implementation details. Therefore, it's not correct to assume that the sub-optimal solutions can be identified and managed from the beginning. Examples of ATD are visible in [42].

7.2.2 Previous research on ATD

The term *Technical Debt* (TD) has been first coined at OOPSLA by W. Cunningham [32] to describe a situation in which developers take decisions that bring short-term benefits but cause long-term detriment of the software. The term has recently been further studied and elaborated in research: in 2013 Tom et al. [33] conducted an exploratory case study technique that involves multi-vocal literature review, supplemented by interviews, in order to draw a first categorization of TD and the principal causes and effects. In such paper we can find the first mentioning of Architectural Technical Debt (ATD, categorized together with Design Debt). A further classification can be found in Kruchten et al. [38], where ATD is regarded as the most challenging TD to be uncovered since there is a lack of research and tool support in practice. Finally, ATD has been further recognized in a recent systematic mapping [31] on TD. Such recent research highlights the gap in the current scientific knowledge, which gives us the motivation for this work.

7.2.3 Previous research on management of TD

Some studies have been conducted on the management of TD, also supported by a dedicated workshop (MTD), usually co-located with premium conferences, such as ICSE and ICSME.

A first roadmap has been created in 2010 by Brown et al. [34]. In 2011 Guo et al. proposed an initial portfolio approach with the creation of TD *items*. The same authors proposed a further empirical study on tracking TD [35] Seaman et al. identified the theoretical importance of TD as risk assessment tool in decision making [36]. TD has also been used for defining part of a method for assessing software quality, SQALE [37]. Such model has been also implemented in a tool, but the main support is currently given on a source code level (very limited on the ATD aspect).

7.2.4 Models for technical debt

The studies in TD are quite recent, and the subject is not mature. Some models, empirical [39] or theoretical [40] have been proposed in order to map the metaphor to concrete entities in software development. We use, in this paper, a conceptual model comprehending the main components of TD:

7.2.4.1 Debt

The debt is regarded as the actual technical issue. Related to the ATD in particular, we consider the ATD item as a specific instance of the implementation that is sub-optimal with respect to the intended architecture to fulfill the business goals. For example, a possible ATD item is a dependency between components that is not allowed by the architectural description or principles defined by the architects. Such dependency might be considered sub-optimal with respect to the *modularity* quality attribute [41], which in turn might be important for the business when a component needs to be replaced in order to allow the development of new features.

7.2.4.2 *Principal*

It's considered the cost for refactoring the specific TD item. In the example case explained before, in which an architectural dependency violation is present in the implementation, the principal is the cost for reworking the source code in order to have the dependency removed and the components not being dependent from each other.

7.2.4.3 *Interest*

A sub-optimal architectural solution (ATD) might cause several effects (for example, as described in [42]), which have an impact on the system, on the development process or even on the customer. For example, having a large number of dependencies between a large amount of components might lead to a big testing effort (which might represent only a part of the whole interest in this case) due to the spread of changes. Such effect might be paid when the features delivered are delayed because of the extra time involved during continuous integration. In this paper, we treat accumulation and refactoring of ATD as including both the principal and the interest.

7.2.5 *The time perspective*

The concept of TD is strongly related to time. Contrarily to having absolute quality models, the TD theoretical framework instantiates a relationship between the cost and the impact of a single sub-optimal solutions. In particular, the metaphor stresses the short-term gain given by a sub-optimal solution against the long-term one considered optimal. Time wise, the TD metaphor is considered useful for estimating if a technical solution is actually sub-optimal or might be optimal from the business point of view. Such risk management practice is also very important in the everyday work of software architects, as mentioned in Kructhen [43] and Martini et al. [44]. Although research has been done on how to take decision on architecture development (such as ATAM, ALMA, etc. [45]), there is no empirical research about how sub-optimal architectural solutions (ATD) are accumulated over time and how they can be continuously managed.

7.3 RESEARCH DESIGN

We preformed 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 many large companies (multiple-case), in order to increase the degree of source triangulation [63]. The reasons for conducting an embedded study (including more than one case within the same context, see company C in the "Case description" section) was to maximize the number of evidences and to strengthen the internal validity of the results with respect to the chosen company.

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 defined by the key characteristics 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 30. In such picture, we show phases of data collection (black boxes), data analysis (white rectangles), results (grey ellipses) and the variable used for axial coding (*Relationship over time*). The arrows, where not explicitly specified, represent the flow of information from an activity to a specific result or vice-versa.

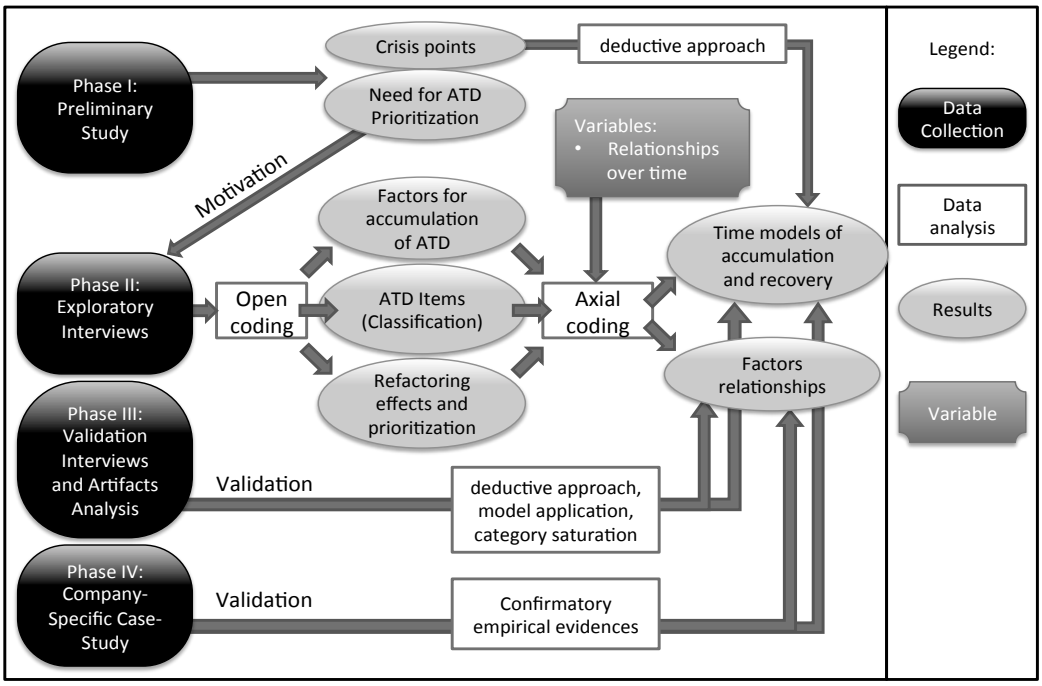


Figure 30 Our Research design

We conducted a first explorative study (phase I), followed by retrospective sessions (phase II), a set of cross-company evaluative interviews (phase III) and finally a follow-up case-specific study (phase IV). The retrospective sessions started by the most effortful events faced by the companies in the recent past and tracking them to the source of the problem, which in some cases would lead to ATD items accumulated in the system. Since retrospective studies usually provide, as a results of the data collection, more emphasis on the challenges currently faced by the interviewees, we included organizations being in different phases of the development, which would experience different occurrences of ATD instances. For example, one organization was currently experiencing a crisis, while others where in the middle of big refactorings and others where mainly focused in feature development.

7.3.1 Case Description

7.3.1.1 Companies description

Company A carried out part of the development 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. The internal releases were short but needed to be aligned, for integration purposes, with the stage gate release model (several months).

In company B, 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 (several months), but the company is in transition towards reducing the release time.

Customers of Company C 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 and some with focus on the platform by different products. 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, while the other units (C1 and C2) teams were mostly co-located.

Company D is a manufacturer of a product line of embedded devices. The organization is divided in teams working in parallel and using SCRUM. The organization has also adopted principles of software product line engineering, such as the employment of a reference architecture. Also in this case, the hardware cycle has an influence

Company E is a company developing software for calculating optimized solutions. The software is not deployed in embedded systems. The company has employed SCRUM 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.

7.3.1.2 Commonalities and differences in the studied companies

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. Some companies use a Domain Specific Language (DSL) to generate part of their source code.

All the companies have employed SCRUM, and have a (internal) release cycle based on the one recommended in SCRUM. However, the embedded companies (A-D) depend on the hardware release cycles, which influence the time for the final integration before the releases. Therefore, some of the teams have internal, short releases and external releases according to the overall product development.

7.3.2 Data collection

We have employed a 4-phase investigation of the ATD items and effects. The four phases (black boxes) and their results (the outcome of the phases, highlighted with arrows connected to elliptic boxes) are visible in Figure 30. Table 21 shows several properties of our data collection: for each phase (explained in details below), the total number of participants, the total amount of hours recorded from the interviews, the number of sessions, if the sessions were company-specific or cross-company and which roles were part of the investigation.

We have conducted a case-study following the guidelines in [63]. We relied especially on semi-structured interviews supported, when possible, by existing architecture documentation in order to provide multiple sources of evidence. Interviews with the architects assured the best representation of the desired architecture for the given case, since the existing documentation was usually not updated or not sufficient for analyzing if ATD was in place. Causes and effects related to the ATD also needed to be reported by the employees directly involved in the case studied. We have asked about possible archival data to analyze, for example source code commits and project data, but we have not found reliable measures to track the extra-effort (interest) related to the studied ATD. Nevertheless, we have surveyed many different roles involved in the studied cases, in order to be able to compare different perspectives on the same case and to mitigate the bias related to only one reporting person.

We have not followed the Grounded Theory strategy for data saturation [55], for two main reasons: we did not aim at a complete saturation of the categories, since we assumed that we might have encountered different situations in different settings and we might have never reached such status. The second reason was because we could not afford to reach complete saturation from a resources point of view, since the interviews

conceded by the companies were too limited to be used for complete saturation. Nevertheless, we have used the principle discussed by Yin in [62], in which the balance between resources and findings is set when the researchers obtain confirmatory evidence of the findings. We report, in the discussion session, a table in which we show which results can be considered confirmed and which ones need further investigation.

Phase I - We started with 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 at the different sites involving several roles. The combined interviews lasted 4 hours and involved developers, testers, architects responsible for different levels of architecture (from low level patterns to high level components) and product managers. The results from the first iteration were evaluated and discussed in a final one-day workshop involving 40 representatives from all the 7 cases (see Table 21).

The preliminary study showed a major challenge in managing Architectural Technical Debt (ATD) and its economical implications related to costs and time. 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 30). During such investigation, we also collected several instances of what was recognized as *"crisis points"*: the companies described such crisis as occurring almost cyclically, points in time when development was particularly slowed down or even stopped until a refactoring took place. From the data we developed the crisis point model explained in 7.5.1.

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. Such activity was conducted especially for further explanations, follow-up questions and evaluation of the developed models. The main rationale for meeting with architects was that they are the main stakeholders involved in the prioritization of ATD, both for their prioritization with respect to the teams' backlogs, but also for their prioritization "against" the features with the product owners.

Each set of interviews followed a process designed to identify important architecture inconsistencies (ATD) that needed to be tracked because of their impact in decreasing developing time. We started with a plenary session where we briefly introduced what ATD is, using references from several sources included in this paper (e.g. the purpose of tracking ATD, [36]), 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 refactoring was prioritized. In particular, we asked when it was prioritized as important or when it was postponed and not included in the next development plan (in the following, we will refer to these two cases as *high-prioritized* or *low-prioritized* refactoring).

The strength of this technique relies on finding the relevant (more costly) architecture inconsistencies (ATD) by identifying their worst effects first, instead of listing a pool of all the possible inconsistencies and then selecting the relevant ones. We have found no other studies applying such "reverse" technique, which add methodological novelty to the current results.

Table 21 Data collection: in the table is possible to see the various phases, each of which included a number of participants covering different roles, either from one specific company or from several companies. The table also shows how many sessions have been conducted for each phase, and how many total hours the sessions lasted.

Phases of data collection	**Number of participants**	**Total of hours recorded**	**Number of sessions**	**Companies involved in each session**	**Roles involved**
Phase I (Preliminary interviews)	25	12	3	Company-specific	Developers, architects, testers, line managers, Scrum m.
Phase I (Evaluation 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 (Evaluation workshop)	10	3	2	Cross-company	Architects, line managers
Phase III (Evaluation interviews 1)	10	4	1	Cross-company	Architects, product owners
Phase III (Evaluation interviews 2)	12	4	1	Cross-company	Architects, developers, scrum masters
Phase III (Evaluation workshop)	20	4	2	Cross-company	Architects
Phase IV (Detailed case-study)	7	6	3	Company-specific	System architect, software architect, scrum master, developers
Informal interaction	7	NA	NA	Company-specific	Software and system architects

Phase III - The third phase consisted of two evaluation 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 crisis model and we asked, when recognized, to strengthen the model with further concrete and real examples. For example, in one case we could see (but we cannot report the picture for confidentiality reasons) the burn-down chart for feature development "abnormally interrupted" due to the crisis point. We also included, where possible, the analysis of artifacts such as lists of *Technical Issues* or *Architectural Improvement* identified within the company. 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 evaluation step we organized 2 plenary workshops with around 20 participants also from 2 other large companies not previously participating in the study, in order to further strengthen the external validity of the results.

Phase IV - Finally, we followed-up the workshop with a company-specific case study. By studying a concrete example in-depth, we aimed at mapping the factors to a typical example and understanding the relationships among the different factors for the specific context. Such technique is recognized as one of the most effective in qualitative research [62], [63] and is called *pattern matching*. In order to study such case, we set up a 2-hour interview with the system architect, the software architect and the scrum master of the involved team. We then followed-up with some team members, 4 developers together with the scrum master. We specifically studied the context of the case described here as C_2. Rather than showing the model to the participants in advance, we preferred to start the investigation from the narrative of a recent ATD case causing effort. We then proceeded with asking specific questions in order for the researchers to recognize the factors and to understand their relationships, in order to match the previously identified patterns. The difference between this case and the ones collected during Phase II consists of the level of details. In this case, we didn't aim at having a collection of cases but rather at gaining as many details as possible in order to provide a better understanding of the specific case.

7.3.3 *Data analysis*

The workshops were recorded and transcribed. The analysis was conducted following an approach based on Grounded Theory [55].

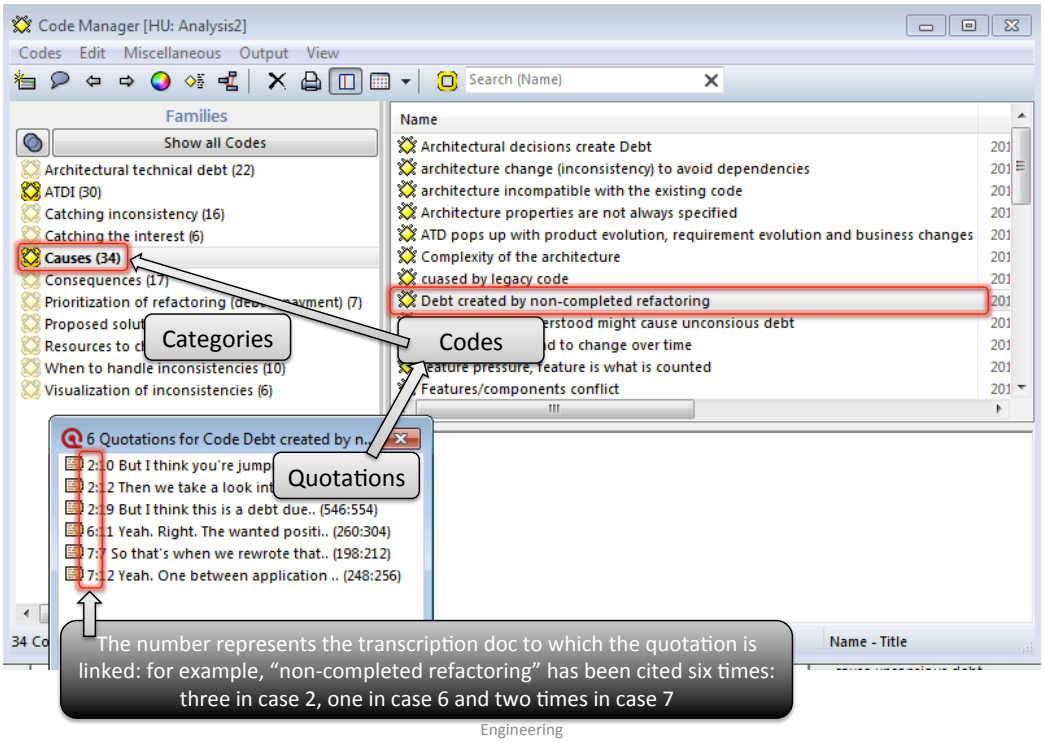


Figure 31 Screenshot of the QDA tool and the chain of evidences

Given the exploratory nature of our study and the need to develop novel concepts and theories about ATD, we opted for using the following methods, frequently employed for the analysis of large amount of semi-structured, qualitative data

representing complex combinations of technical and social factors. The analysis followed the steps highlighted in Figure 30.

Open Coding – In phases I and II (the exploratory ones) the first step was to analyze the data in search for emergent concepts following open coding, which would bring novel insights on the analyzed issue. For example, we did not know in advance that *non-completed refactoring* would lead to new ATD. Without the open coding but just looking for predefined categories would have probably caused us to miss this novel concept. Then we categorized the codes using the taxonomies developed during the pre-study. For example, we used *causes* as a pre-defined category. Codes within the same category were also grouped in order to create concepts: for example, within the *causes* categories we have found 34 codes, which were grouped in higher level codes (or concepts), which are represented in the factors outlined in section 7.4.

We used a Qualitative Data Analysis (QDA) tool for open coding, atlas.ti. Such tool gives support to keep track of the links between categories, codes and the quotations they are grounded to (providing, as recommended in [63] a *chain of evidence*). A screenshot with explanations is reported in Figure 31, showing the elements used in the analysis (*quotations*, *codes* and *categories*) and how they are connected through the tool. The initial codes were 144, of which 97 were selected as relevant for the RQs in this paper. We then merged the redundant codes and filtered them by their *groundedness* (based on how many quotation they were based on), selecting only the ones that had at least 2 quotations (i.e., having *confirmatory evidences*): the final number of codes is 38 [79]. The predefined categories were decided by the three authors altogether. The open coding was performed by the first author, and the codes were checked by the other authors periodically.

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. The aim was to understand relationships (e.g. causality, dependencies, etc.) among the factors. This step showed the presence of sequences or patterns of accumulation factors for ATD over time. In particular, such analysis produced the model for accumulation and refactoring of ATD explained in section 7.5, and the same analysis on the in-depth case study was done to conduct *pattern matching* with evaluation purposes (see section 7.7.2). The axial coding was carried out by the first author supervised by the second author, while the third author checked the results.

For this research activity, we needed a tool for visualizing the factors and the refactoring of ATD over time. We therefore exported the related codes and concepts from Atlas.ti into Microsoft PowerPoint, which allows the easy creation of a graphical timeline. In order to place the factors on the timeline, we used the time information contained in the quotations linked to the factors: for example, the informants mentioned that the uncertainty about the use cases was present in the beginning of the development and that the urgency was present especially when close to a release. Some of the factors, for which there was no specific time but they were mentioned as being always on going, we represented them as spanning from the development start to the release. As for those factors that were not bound to time but were happening in an ad-hoc fashion (for example, non-completed refactoring) were omitted in the model since they were not part of a recurrent pattern. The same approach was used for the concepts related to the prioritization of ATD refactoring: for example, the quotations from the informants usually reported the refactoring as postponed with respect to the release. Therefore we represented the refactoring activities as the decrement of ATD happening after the release point. The outcome of axial coding, the model of accumulation and refactoring over time, is visible in Figure 33.

Deductive Approach – We performed deductive analysis especially for evaluation purposes, but also in order to relate two different models. For example, after the open coding analysis phase, the factors and the models were deductively checked against the overall model of crisis point, developed and evaluated 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 7.5.2.

In particular, we projected the model showed in Figure 33 over a longer time span, taking three cases with respect to three different refactoring strategies: when all the ATD is removed, when ATD is partially removed and when is not removed at all (Figure 34, Figure 36, Figure 37). Then we have related these three options with respect to the crisis point model in Figure 32. The outcome of the projection combined with the crisis point model led to the hypothesis described in Figure 38, where different refactoring strategies would lead to different crisis points.

The deductive approach was also used during interviews in the evaluation phase III, when we used the models for eliciting concrete cases to confirm (or reject) the inductively obtained hypothesis made during phases I and II. An example can be found in section 7.7.2.1, where we report concrete quotes answering the evaluation questions on the Crisis Model.

In phase IV, the deductive approach was used during data collection, when we were asking questions in order to recognize, in the in-depth case study, the factors found in phase I and II. More specifically, we have used the *pattern matching* analytical strategy recommended in [62]. Such strategy is used to verify the existence of previously formulated patterns (hypotheses) after a new collection of data. In our cases, we used the models obtained by the data collection conducted during phase I and II as *pattern tests* to be used during phase IV to reveal the pattern. This kind of evidence contributed in evaluating the formulated models (with different strength for different results, as explained in 7.7.2). In this case we used a similar practical approach used for Axial Coding, placing the factors in a timeline. This way, we had two similar (with the same factors over time) models to compare, one from the inductive analysis of phases I and II and one from the case study: the comparison would lead to the matching (or not) of the found patterns.

7.3.4 Factors and models evaluation

As explained in Figure 30 we conducted two steps of evaluation (phase III and IV). The crisis point model (outlined in 7.5.1 was developed during phase I. It was qualitatively evaluated during phase II, where *all* the informants recognized the model as representing the facts in their company. During phase III and IV we also probed the crisis model by asking more precise questions on the evidence of such a crisis model. In one case, the participants showed us a burn-down chart for feature development "abnormally interrupted" (citing from the whiteboard) due to the crisis. Unfortunately, we cannot report the actual source for confidentiality reasons.

The models for accumulation and refactoring of ATD were developed in phase II and took as input the deductive model about the crisis point developed previously. The new models were then evaluated during the evaluation workshop of phase II and during phase III.

The factors were evaluated both during phase III and by *pattern matching* during to the in-depth case studied in phase IV. Such technique is recognized as one of the most effective in qualitative research [62], [63].

7.3.5 Models of Accumulation and Refactoring of Architecture Technical Debt

We have divided the results in four parts: first we highlight the causes for ATD accumulation (factors). Then we use such factors to describe a model for accumulation and refactoring of ATD over time. We then narrate in a chronological sequence of events an additional industrial case, which shows the factors and their relationships over time. We finally show results from the evaluation process of the factors and models, both with quotations and matched pattern.

7.4 CAUSES OF ATD ACCUMULATION (FACTORS)

7.4.1 Business factors

7.4.1.1 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. 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.

7.4.1.2 Uncertainty of use cases in early stages

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 sprints).

7.4.1.3 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 low-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.

7.4.1.4 Priority of features over product architecture

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

7.4.1.5 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.

7.4.2 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.

7.4.3 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.

7.4.4 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.

7.4.5 Uncertainty of impact

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 exists some ATD that remains *unknown*.

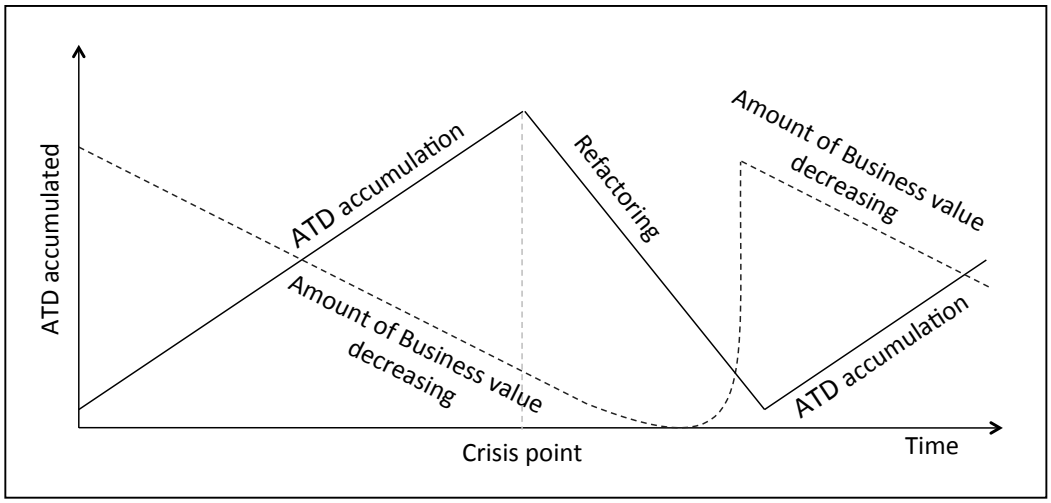


Figure 32 Crisis point model

7.4.6 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.

7.4.7 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.

7.4.8 Lack of knowledge

Software engineering is also an individual activity and the causes for ATD accumulation can also be related to sub-optimal decision taken by individual employees due to:

7.4.8.1 Inexperience

New employees are more subjected to accumulating ATD due to the natural non-complete understanding of the architecture and patterns.

7.4.8.2 Lack of domain knowledge

This factor might be related to the previous one, or, as the informants mentioned, to the generalization of Agile teams, which might need to develop a feature accessing a complex component of which they don't have expertize.

7.4.8.3 Ignorance.

Lack of knowledge about where the architectural rules are stored (documentation).

7.4.8.4 Carelessness

Lack of awareness of the importance of architecture. A recurrent statement from the informant is that having documentation is not enough to avoid having architecture violations.

7.5 ATD ACCUMULATION AND REFACTORING 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.

7.5.1 Crisis-based ATD management

The current management of ATD is driven by a crisis (Figure 32). 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 event with limited monitoring and limited reduction of ATD growth during development. In fact, the long-term improvement is considered risky invested time.

7.5.2 ATD accumulation and refactoring trends during feature development

In Figure 33 we present the various phases of ATD accumulation over time, on the left part of the graph, and the hypothetical refactoring ("complete refactoring") of ATD on the right, divided by different kinds of identified ATD.

7.5.2.1 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 Figure 33). Such part is composed by several components described previously in section 7.4: *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 *technology evolution*). Some of them might be considered as multipliers for the other kinds of ATD, but for simplicity and for lack of more precise measures, we treat them as constants in the graph.

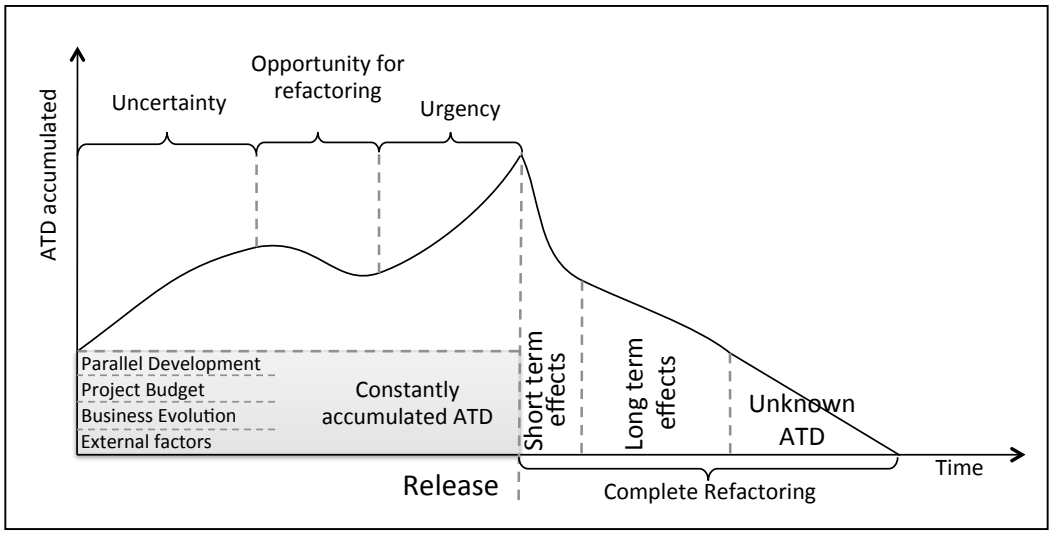


Figure 33 Factors of constantly accumulated ATD, phases and kinds of refactorings

7.5.3 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 7.4.1.1 and 7.4.5), the curve on the graph in Figure 33 representing ATD accumulation tends to raise in the beginning until the team has a clearer understanding of the requirements, desired design and desired architecture altogether. At this point, the hypothesis is that ATD accumulation would slow down. The ATD starts again 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 refactors part of the ATD needed to deliver the feature, but it's unlikely that all the accumulated ATD is refactored during this phase (especially the constant one). However, this seems to be a good *opportunity for refactoring* in the process when the team might decide to take care of the ATD before the release. The *project budget* factor (7.4.1.5) might have a negative impact on such practice though, demotivating the team to look for such opportunities.

7.5.3.1 Refactoring and its prioritization

Once the feature is released, there is ATD left in the system. The ideal case is that the ATD would be 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 *uncertainty of impact* in section 7.4.5), 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 clear business need of it and because such ATD can be refactored by being included in the successive feature development. Examples of such short term improvements include small or local (not spread out in the whole system) adjustments of patterns to allow a feature to be implemented. 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 higher level of abstraction that has not been implemented during the development of the features. 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 would not make sense to include the refactoring 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 (as explained in section 7.4.1.4).

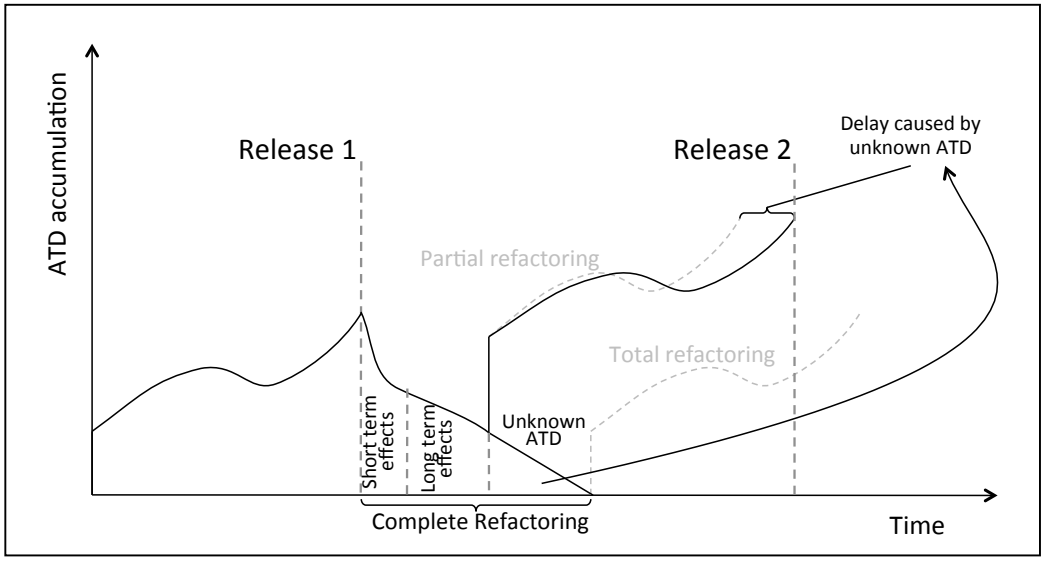


Figure 34 Refactoring maximization: some ATD is always accumulated and impacts next development (e.g. release 2). Total Refactoring is not realistic in practice. The best option is Partial Refactoring, when short-term and long-term ATD is removed.

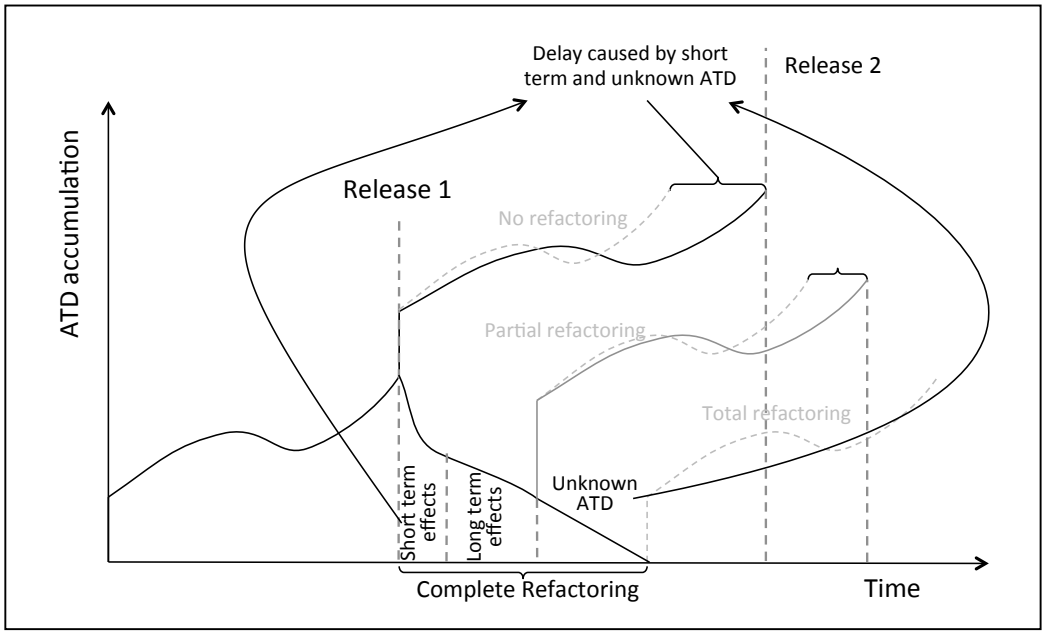


Figure 36 Refactoring minimization: when No refactoring is applied, both short-term and unknown ATD would impact release 2. However, there might be a short-term benefit with respect to doing long-term refactoring (difference with Partial refactoring)

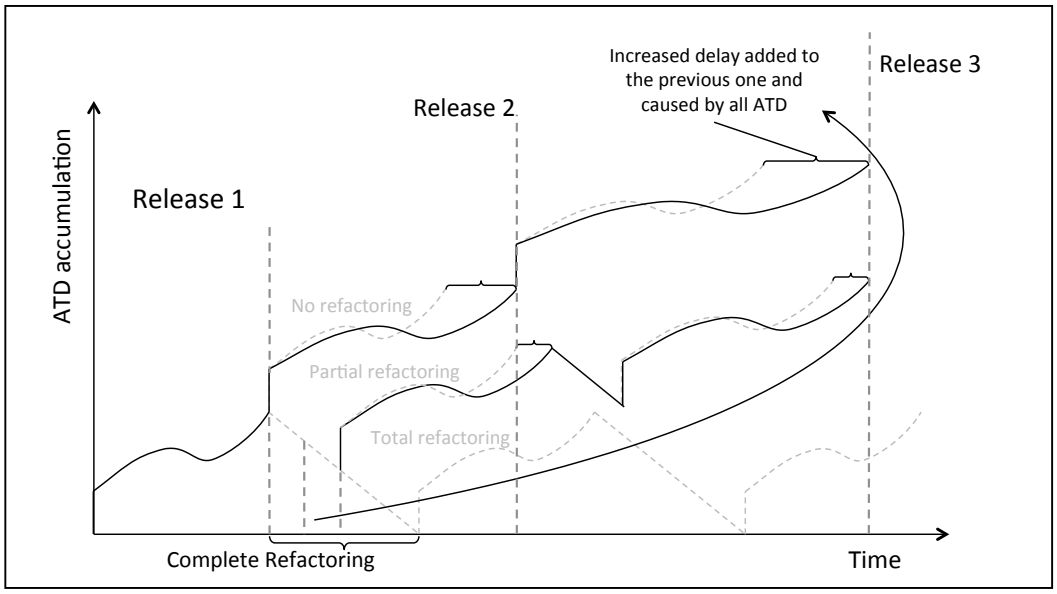


Figure 37 Comparison of refactoring strategies: after a number of features released (symbolically 3 in the picture), the long-term ATD starts to have an impact, decreasing the advantages of the short-term benefits given by the No refactoring strategy.

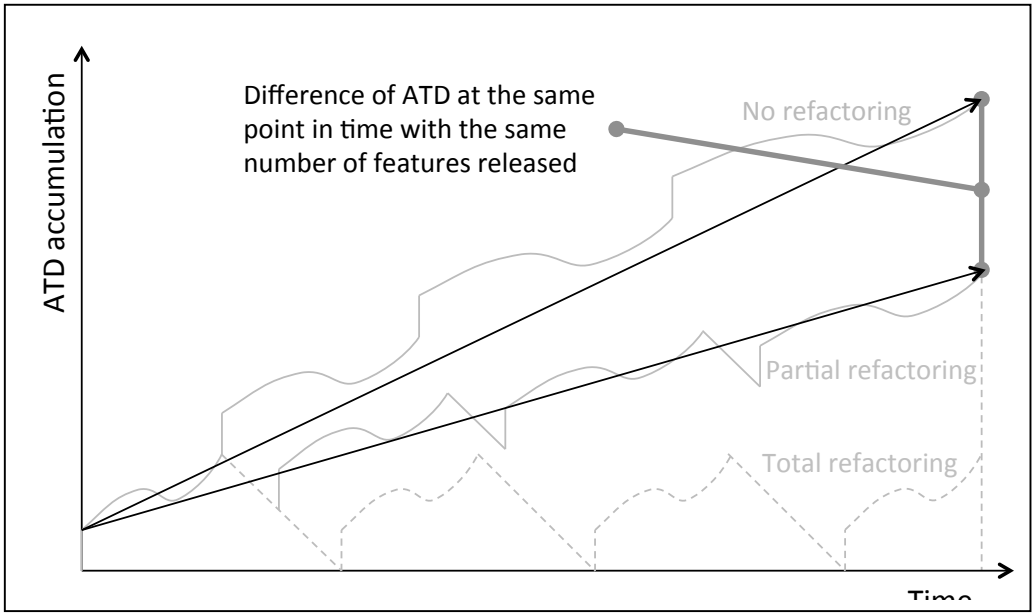


Figure 38. Comparison of refactoring strategies: after a number of features released (symbolically 3 in the picture), the time elapsed to deliver would be the same but the ATD present in the system would be more in the No recovery strategy, which will affect long-term development

7.5.4 Comparison of refactoring strategies

We will show what implications, in terms of refactoring strategies, can be drawn from the current results. We show this by analyzing the combination of the models previously shown and how they lead to different outcomes (Figure 34-Figure 38), with respect to different refactoring strategies. In particular, we show how projecting several instances of the *phase model* over time and combining such projection with two different refactoring strategies, *refactoring maximization* and *refactoring minimization*, would bring different outcomes in term of development crises.

Refactoring maximization: in Figure 34 is represented the scenario in which both short-term and long-term ATD are refactored. We can see how the constant ATD and the delay effect from the *unknown* ATD are continuously accumulated, making the

accumulation strictly monotone even when the refactoring is maximized. This would exclude the possibility, for the companies, to apply a *complete refactoring* strategy, in which all the ATD is removed. The maximization of refactoring consists of, for the company, adopting a *partial refactoring* strategy.

Refactoring minimization: in Figure 36 we show the case in which a *no refactoring* strategy is applied in contrast with the previously mentioned *partial refactoring*. Even though the feature might be released earlier with respect to *partial refactoring* (when all possible refactoring was performed), the delay caused by the ATD grows because of the addition of *unknown* and *short-term* ATD.

Short-term comparison of strategies: in Figure 37 we have represented a further projection of both strategies for a hypothetical number of 3 releases (the same effect might be reached with a different number of releases, depending on the context): in the *no refactoring* strategy we can see the increasing of the delay caused by the ATD with *long-term effect* with respect to the *partial refactoring* function. In conclusion, after some time *partial refactoring* might be as convenient (in terms of features released) as *no refactoring*, with the difference that in the second case there is more ATD in the system (Figure 38).

Long-term comparison of strategies: By projecting the same trend for longer time and combining it with the *crisis point* model (assuming that the same crisis will happen when the same amount of ATD is accumulated), we can see that with a *partial refactoring* strategy the crisis point would be delayed (Figure 39).

Choice of the optimal refactoring strategy with respect to the crisis points: taking in consideration the crisis points perspective, the main choice for the companies is to balance the prioritization of refactoring in order to have as less number of crisis points as possible, or to delay the crisis point over time according to the lifecycle of the product. The evidences collected suggest that the best strategy to avoid development crises is the partial refactoring.

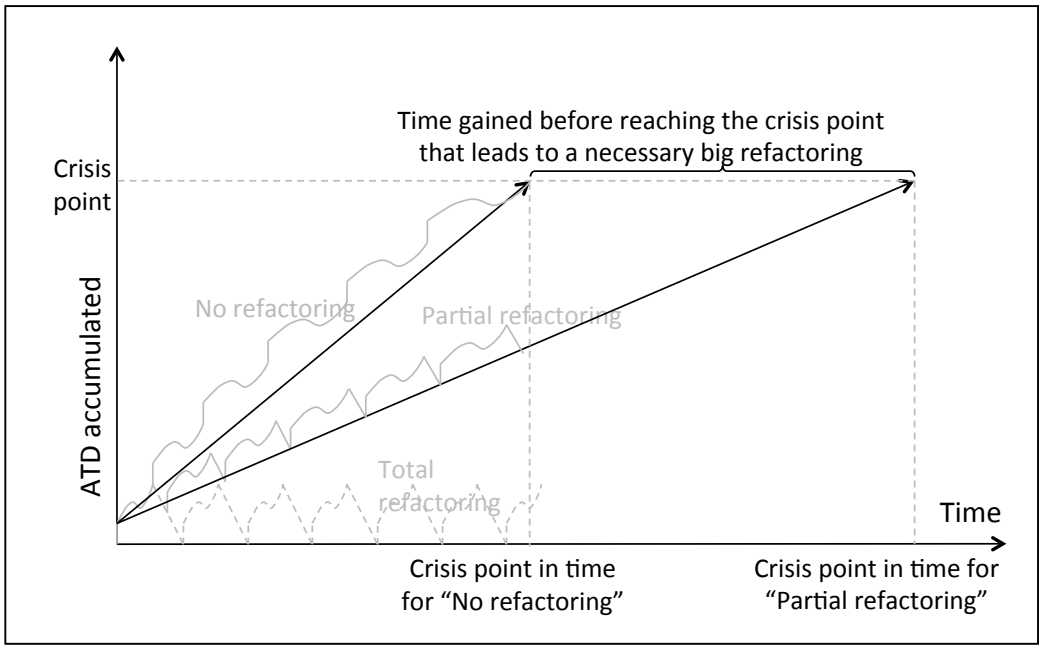


Figure 39. Long-term comparison of the strategies: given the trend of inevitable accumulation of ATD, a crisis point will eventually be reached in both cases. The gain for the company is to reduce the number of crises and therefore the number of costly refactorings.

7.6 DETAILED CASE-STUDY

We conducted a follow-up, in-depth case-study with one of the cases involved previously in the investigation. Specifically, we studied the case C_2. We investigated one of the recent refactorings conducted at the site in order to find patterns that would evaluate our previous hypotheses (as recommended by Yin and Runeson and Höst, [62], [63]).

Since we wanted to show the relationships among the factors over time, we will describe the case in a chronological fashion. We highlight key events in the timeline and then we describe what happened between the two events. We will focus only on the part of the system that was studied during the interviews.

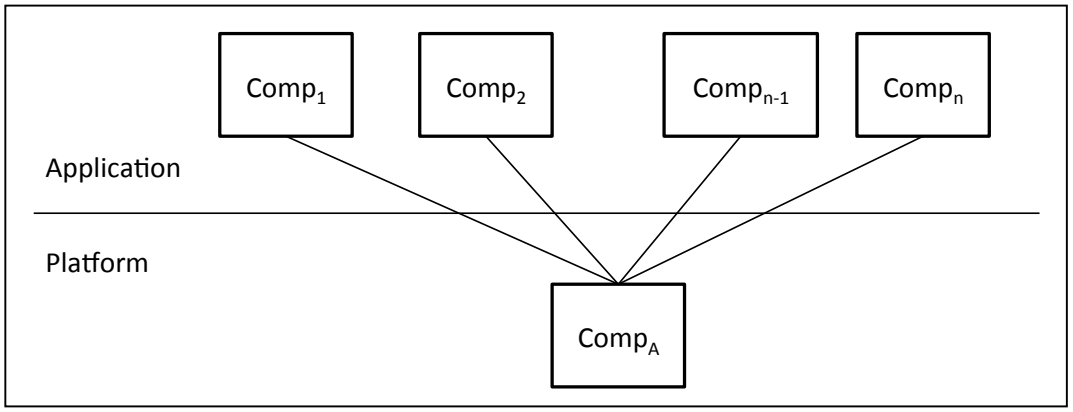


Figure 40 Simplified architecture layers (Platform- and Application-layer) of the studied system

7.6.1 Chronological narrative of events

T_0: start of development: a new product of the product-line was requested by the market and by specific customers. Such product would include a set of features, but would also need to be integrated with the previous platform. Within the product, one of the components (that we call *Comp$_A$*) was integrated in the platform since it contained functionalities shared by to other components (that we call *Comp$_{1...n}$*) included in different applications. $Comp_A$, being a central component, had (legal) dependencies with the other components, as visible in Figure 40.

$T_{RefCompA}$: discover of the need of refactoring. After the integration of $Comp_A$ and after the component had been in use within the product for around 3 years, the system responsible found that there was an anomaly in the number of defects reported in the system, which led to a crisis. After the analysis of the bug reports, it was possible to understand that the design of $Comp_A$ was the cause for such anomaly (many problems were tracked back to $Comp_A$). Given the situation, the system architect and the managers decided to prioritize the refactoring of $Comp_A$. Such refactoring involved the restructure of $Comp_A$, considered not well modularized and too complex. The estimation for $Comp_A$ was considered as *RefA*. $Comp_A$ was therefore duplicated (branched) in $Comp_A$', in order to be refactored in parallel to the continuous development of features. It was not possible, obviously, to stop the use of the product(s) using $Comp_A$ and its development during the refactoring, since the overall re-design of $Comp_A$ required a substantial amount of time (around 6 months).

$T_{RefComp1...k}$: during the estimation for Ref_A, however, the impact of the changes implemented in $Comp_A$ with respect to $Comp_{1...n}$ was not correctly understood and estimated. After the refactoring of $Comp_A$' started, it became clear that the APIs needed to be updated, and the work needed to refactor $Comp_{1...n}$ according to the changes in $Comp_A$' required more time than expected. The refactoring of $Comp_{1...n}$ was therefore estimated and broken down to stories that would need to be prioritized in the backlog of the teams allocated to such work. However, not all the refactoring work could start

in all $Comp_{1...n}$ before the changes in $Comp_A$' would be completed. Therefore, the refactoring was initiated in $Comp_{1...k}$ but not in $Comp_{k+1...n}$.

T_{NewReq}: New application requirements involving $Comp_{k+1}$…$Comp_n$. While the refactoring of $Comp_A$' was still on-going, new feature requirements were received by the teams developing $Comp_{k+1...n}$. The backlogs were re-prioritized and the new stories were given a priority higher than the priority of the refactoring stories. Such decision was also due to the fact that $Comp_A$' was not ready at such moment. However, the stories for coordinating the refactoring with $Comp_A$' remained low-prioritized also when it became possible to start.

$T_{NewComp}$: New applications. Starting during this time, new features were requested by the market and the customers. Therefore, new application components were created ($Comp_{n+1...m}$). Such components were designed to interact with $Comp_A$', since such component was being refactored and was considered more updated and better usable by the new application due to its re-design.

$T_{NoRefCompk+1...n}$: When the refactoring of $Comp_A$' was completed, the refactoring of $Comp_{k+1...n}$ were still not carried out. Despite the fact that the applications were using the old $Comp_A$, such motivation was not enough to high-prioritize the stories for refactoring. From this point on, the system was using a duplicated component ($Comp_A$ and $Comp_A$'), in which part of the applications were interacting with $Comp_A$ and part of them were interacting with $Comp_A$', causing a duplication of maintenance work.

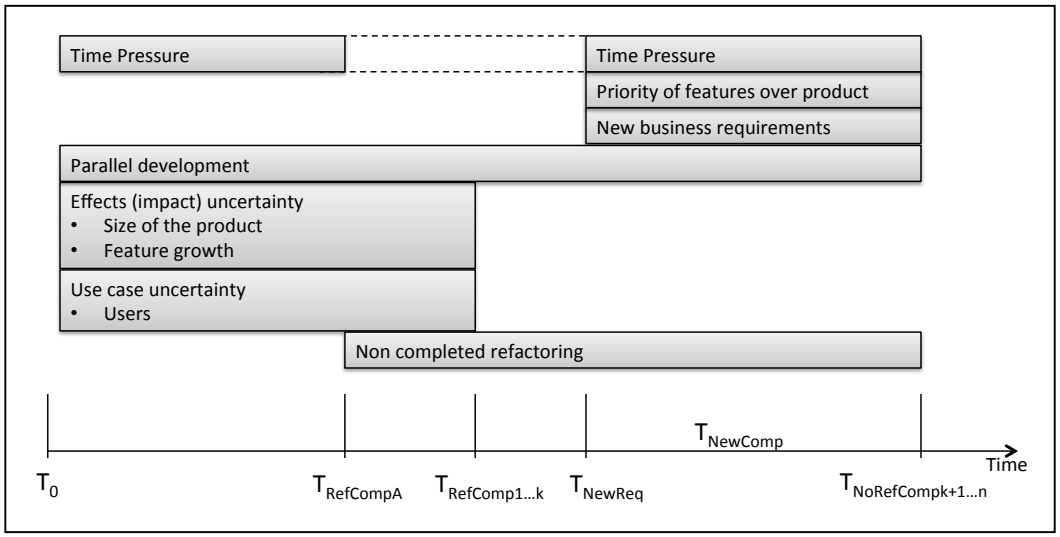


Figure 41 Timeline of the case-study in relation with the factors influencing the accumulation of ATD over time

7.7 EVALUATION

To evaluate the exploratory results collected in phase I and II, we conducted and analyzed interviews (phase III) and the in-depth case-study (phase IV). In this section we show how the factors and the models where evaluated by the two approaches.

7.7.1 Factors evaluation

We deductively analyzed the case-study in order to recognize the factors previously obtained from the multiple case study.

Figure 41 shows a time-line with the major events explained in the previous section. Part of the events cannot be referred to a single point in time, but rather to a time interval between two time points. We mapped the factors to the time-line, in order to

visualize their relationships over time (used in the next section for visualizing pattern matching). The following factors were recognized:

- **Uncertainty of use case in early stages**: citing the interviewee: "*the team did not completely understand the impact of the developed component on the users of such component*". Such factor was occurring at least in the beginning when the ATD was accumulated, but also in the beginning of the refactoring, where not all the users (stakeholder) of the refactored $Comp_A$ were identified. For example, the interviewee mentioned the lack of awareness about part of the testers interacting with $Comp_A$.

- **Uncertainty of impact**: citing the system architect: "*at time [$T_{RefCompA}$] it was easy to think that the issues [ATD] could be solved by just refactoring [$Comp_A$]*". Clearly the team did not understand the impact of introducing ATD into the system, affecting $Comp_{1...n}$ and therefore ignoring the impact of changes.

- **Parallel development:** this factor is clearly present and influencing the accumulation of ATD for the whole period. First of all, the product was developed in parallel to the other products, which contributed to "isolate" the development from the users of $Comp_A$ and the development of $Comp_{1...n}$. Such isolation contributed to the *uncertainty*.
  After the refactoring of $Comp_A$ was started at $T_{RefCompA}$, the parallel backlogs caused a misalignment in the prioritization of the distributed stories related to completing the refactoring, which led to the duplication of component $Comp_A$ (accumulation of ATD through uncompleted refactoring).

- **Time pressure**: the need for delivering quickly (short lead time) was mentioned especially in the beginning before the reaching of the crisis point $T_{RefCompA}$, and during T_{NewReq}, after the new requirements came in and new stories were added to the distributed backlogs. In the middle of the analyzed time-span, the refactoring was started, so it could be easy to think that the time pressure decreased. However, the reaching of the crisis triggered the refactoring, which was started in order to avoid the time spent on fixing a large amount of bugs instead of developing new features. We can see then how the time pressure played a role in that period as well.

- **New business requirements**: new requirements were obviously received at T_0, when the product development started, and at T_{NewReq}, when new features and customer-specific products were requested or identified by the product owners.

- **Priority of features over product**: the low-prioritization, during T_{NewReq}, of the stories related to complete the refactoring is a clear sign of this factor, which directly led to the uncompleted refactoring and the duplicated component (ATD).

- **Uncompleted refactoring:** from the start of the refactoring of $Comp_A$, the ATD accumulated through the duplication of the component was constant. However, the ATD (duplication) was not considered as such (but only temporary duplication) until it became clear that the substitution of $Comp_A$ with $Comp_{A'}$ was not possible due to the low-prioritization.

In conclusion, the case study confirmed the presence of many of the factors listed in section 7.4 that influenced the accumulation of ATD over time.

7.7.2 Models evaluation

For each model presented in section 7.5, we show how the evaluation interviews in phase III and the case-study conducted in phase IV brought evidences to the exploratory results of phase I and II. In particular, the distribution of the factors should

be compared with Figure 32 and Figure 33 in order to understand the matching of the patterns.

7.7.2.1 Crisis-based ATD management (7.5.1)

During the evaluation interviews we collected data on the models from company A, B, and the three cases $C_{1,2,3}$. We asked if the crises occurred recently and how the informant experienced them. We list a selection of relevant quotations from different informants all from different companies (not specified for confidentiality reasons). We cannot report all the evaluation data due to space constraints, but the ones listed here represent a good sample of the data used for models evaluation.

Chief architect:

"*When we realized that we were going into a situation where we had so many variants that the monolithic structure wouldn't be able to sustain that feature growth and those types of variance. We actually realized this before we hit the crisis point. It wasn't that we weren't able to deliver projects. We realized it before. So we didn't actually hit the crisis point and people stopped delivering software, but we were pretty close. We could see that the next project will not be able to deliver. So this will be a more expensive project. We have the luck that we got buy in from management that we needed to do these changes.*"

System architect:

"*So we had to take resources from feature development to stabilization. We just stopped feature development to stabilize what we had first. Then we could continue with features.*"

Software architect:

"*I think we have exactly the situation you described [referred to the slide showing the description of crisis point]*."

System architect:

"*[] we had to stop introducing new features and just fixing bugs. And then we have also started a redesign of some components. […] And then the management understood that we need to do something about this component, redesign […]*"

System architect:

"*I recognize […], we had a lot of [bug fixing] in the same area. And we couldn't continual fixing here and there with sort of workarounds or not clean fixing. So we realized we needed real refactoring to solve all together and to ease the introduction of new features.*"

From the 5 companies interviewed, it's quite clear that the informants agreed on the validity of the model and they shared several experiences. We can also see confirmatory evidences that the crisis model is valid for the in-depth studied case (Figure 41). The crisis when the ATD was revealed is represented by the high number of defects, which led to allocate a large percentage of development to bug fixing. Such activity decreased the time dedicated to new features, together with the perception of the slowing down of development feature themselves.

7.7.2.2 Constant ATD accumulation (7.5.2.1)

In the studied case we can see how it was never possible to avoid the presence of ATD in the system, even by prioritizing the big refactoring. We could evaluate most of the factors that we have claimed to be the cause of constant accumulation of ATD. In the case study, *parallel development* is easily visible as influencing factor for the whole

time, while *business evolution* is also quite persistent, excluding the phase after the crisis, when the impossibility to develop led to the refactoring.

7.7.2.3 Phases of ATD accumulation (7.5.3)

We can compare the model of the phases for accumulation and refactoring of ATD (Figure 32) with the distribution of the factors in Figure 41 related to the case-study time-line, in order to understand the matching of the patterns.

Matching patterns:

- *Uncertainty of use cases* in early stages is confirmed as a factor boosting the accumulation of ATD.
- The case suggests how the *uncertainty of impact* would also be extended until the actual refactoring was started.
- *Parallel development* was impacting the accumulation of ATD. The news is, parallel development had also an impact on refactoring.
- The "inflow" of new business requirements (*business evolution*) influenced the accumulation both in the beginning of the development and also when new features were added. Before the second wave of requirements, there was a time span when the refactoring was started. However, we cannot map this decrement of the ATD with the refactoring opportunity hypothesized in Figure 33 since the main reason for such refactoring was not the lack of time pressure, but rather the need to fix the bugs (and the source of them).
- Time pressure is present throughout the whole time in the case-study. Although the refactoring has been prioritized, such decision was taken because the current development was so slowed down that it was too difficult to add new features. The refactoring was therefore also prioritized in order to improve lead time (time pressure). Therefore, time pressure is to be considered as a constant accumulator for ATD.
- Urgency, in Figure 33, was present close to the release, but it's just part of the overall time pressure, which is spread throughout the whole process. In order to decrease the lead time, management focuses on different target in a reactive way: first development, then refactoring when development starts to become difficult (approaching of the crisis).

In conclusion, most of the accumulation phases and components of the model have been confirmed, even though their spanning might vary and some components could not be confirmed by this specific case study: uncertainty is more present in the early stages but might be quite protracted, time pressure is constant throughout the development, as well as parallel development and business evolution. The refactoring opportunity (visible in Figure 33) needs to be further confirmed: in the specific case study, the refactoring was high-prioritized for the same reasons why it was low-prioritized in other phases, i.e. for time pressure.

7.7.2.4 Refactoring prioritization strategies (7.5.3.1 and 7.5.4)

The case-study is a good example of how the long-term refactoring was prioritized. Only after a crisis the refactoring was reactively started, exactly as shown in Figure 32. We cannot compare this chain of events with an hypothetical case where the long-term refactoring would have been prioritized in advance, but we can see how the project reached a crisis, and from the issues reported it was possible to identify the presence of long-term ATD as a source of the crisis. We can therefore infer, from the evidence, that the *No refactoring* strategy led to the crisis, but we don't know how faster than if the debt would have not been taken or refactored earlier. Once there, postponing

architectural refactoring (long-term) resulted quite costly and was not completed, leading to accumulation of different ATD. Therefore, the strategy of postponing long-term ATD refactoring after the crisis seems to be a non-optimal solution. We can consider these data as partially confirming the previous models, considering though the need for further evidences from more cases in this direction.

Table 22 Exploratory and evaluated results

Results	Confirmatory evidences from multiple interviews	Confirmatory evidence from case-study	Exploratory evidence from multiple interviews	Exploratory evidence from case-study
Factors influencing ATD accumulation (RQ1): • Uncertainty of use-case in early stages	X	X		
• Business evolution	X	X		
• Time pressure	X	X		
• Priority of features over product	X	X		
• Split of budget	X			
• Design and Architecture documentation	X	X		
• Reuse of Legacy / third party / open source components	X			
• Parallel development	X	X		
• Uncertainty of impacts	X	X		
• Non-completed refactoring	X	X		
• Technology evolution	X			
• Lack of knowledge	X	X		
Models of ATD accumulation and refactoring (RQ2) • Crisis-based ATD management	X	X		
• Phases of ATD accumulation	(X)	X	X	
Refactoring strategies (RQ3) • *Partial refactoring* is the best option for the maximization of refactoring. *Complete is refactoring* not realistic.	(X)		X	X
• Drastic minimization of refactorings (*No refactoring*) leads to development crises often in the long run.	(X)		X	X

7.8 DISCUSSION

We have conducted an exploratory multiple-case study in 7 large software organizations, showing factors causing the accumulation of ATD (RQ1), trends in such accumulation and refactoring over time (RQ2) and analyzing different outcomes (in terms of development crises) for different refactoring strategies (RQ3). We have backed up the exploratory results, collected in 2 phases, with cross-company evaluation interviews and an in-depth case study about a concrete ATD case, which strengthened the results.

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 refactoring. 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 or through an in-depth case-study, but that need to be quantitative complemented and assessed in the future.

7.8.1 Implications for research

The investigation in this paper brought to light several results throughout different phases, especially during the exploratory phases I and II. Some of such results were evaluated through phases III and IV, where we run interviews and we conducted an additional, evaluative case study. We add an evaluation table (Table 22), were we can see a summary of the results and by which means they have been evaluated.

These results represents a first step towards the development of middle-range theories [66], which are not considered unversally valid, but only within a range of contexts. We investigated the results in 7 similar sites, which allows us to claim the validity of our findings to the context of large software companies developing embedded software and employing ASD. Another attribute of the context is the geographical location of the companies, all placed in Scandinavia. In different geographical areas the results might be different. We therefore encourage further investigation of the results in other contexts in order to make another step towards further generalizing the results (as explained in the external validity section).

Answering the RQs contributes to the body of knowledge, according to the gaps identified in a recent systematic mapping study [31], by providing a reliable industrial study based on the experiences of several ATD stakeholders: architects, developers and product owners. In particular:

- RQ1: we have studied the socio-technical factors related to phenomena of accumulation and refactoring of ATD. These factors might be studied and treated separately. Such factors offer a better understanding of the overall phenomenon of ATD accumulation and refactoring, which is a prerequisite for its management.
- RQ2: we have developed and evaluated two qualitative models of the trends in accumulation and refactoring of ATD over time based on its prioritization: the crisis model and the phase model.
- RQ3: we have provided constraints and recommendations about different refactoring strategies and their effects on development crises. Such recommendations should be tested with further case studies.

7.8.2 ATD and software architecture management

As we can see from

Figure 33, 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 monotone. These results show two main findings important for the software architecture community: that it's quite likely that the initial software architecture would change because of changing requirements and that a number of sub-optimal solutions will always been implemented because of the several factors causing ATD. Therefore, such drifting needs to be managed continuously, especially to uncover the ATD that it's unkonwn and in order to prioritize the ATD that have the worse effects over time. It's important to understand how to continuously analyze the architecture in order to develop better solutions and apply a suitable refactoring strategy.

7.8.2.1 ATD and Agile Software Development

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. Another trend related to ASD is having teams that have to focus on a feature and are free to touch any component. The lack of domain knowledge, however, boosts the accumulation of ATD.

From this investigation we can see how the chosen strategy used for ATD prioritization and management would have an impact on ASD: frequent crisis points might nullify the responsiveness and continuous delivery achieved by the Agile practices. 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. The results contribute to the current body of knowledge according to [31], where the authors claim the need to study

7.8.3 Implications for practice

The results have implications for multiple roles in the organizations: from product managers and product owners, to architects and developers.

7.8.3.1 Implications for product managers and product owners

The goal for software companies is to avoid development crises, since platform creation and/or huge architectural refactorings are long and costly activities and would stop the continuous delivery of features to the customer. We have analyzed different refactoring strategies with respect to the minimization of crises. The evidence suggest that a *complete refactoring* strategy is not possible, and therefore that the main goal for a software company cannot be to have an "eternal" system, but rather to reduce the number of times the development crises are repeated over time before the retirement of the product. This means that a company needs to proactively adopt a partial refactoring strategy and needs to plan refactoring activities as part of the product development.

Some of the factors have a disincentive (or incentive) effect on ATD accumulation. The known disincentives are the ones described in section 7.4.1 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. An important incentive, especially relevant for product owners, is the prioritization of ATD refactorings, especially the long-term ones.

7.8.3.2 Implications for architects and developers

In many cases, the model in Figure 32 has been found valid to describe the current events. Automatic static analysis tools together with the current practices of lightly documenting software architecture don't seem to be able to provide enough awareness of the ATD present in the system, if not on an intuitive level. Therefore, more continuous analysis needs to be done in order to assess the current ATD in the system and its impact (interest). *Uncertainty of impact* might be decreased by identifying the debt, localizing it and understanding the stakeholders involved in the payment of the interest. Such practice might be supported by various metrics (usually context

dependent), but the developers, architects and the involved stakeholder need to work together for the aggregation of information and the communication of the risk incurred in taking debt that has a wide impact on the organization.

7.8.4 Limitations

The graphs in this paper (Figure 34, Figure 36, Figure 37, Figure 39 and Figure 38) 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, and their relationship with time. The results are qualitatively developed through a thorough research process including several triangulation techniques recommended [63] for qualitative studies: we used a wide amount of qualitative data coming from many informants throughout four iterations of investigation (see Table 21), from seven sites and with different roles, which allowed us to compare and test statements among themselves. Furthermore, architecture documentation and improvements backlogs have been evaluated as secondary data. This has allowed us to apply source triangulation. We have also applied three ways of qualitative evaluation of the results: plenary workshops, interviews and a company-specific case-study. Table 22 shows which results have been consolidated with which method, and which are, on the other hand, still exploratory and need further evaluation. Another limitation is the use of a single, company-specific case-study: due to resource constraints, the authors could not perform the same in-depth investigation in other organizations, but we are confident that the whole research community, especially the empirical one, will contribute to this topic with more evidence in order to build an even more solid body of knowledge on the ATD management.

7.8.5 Future work

7.8.5.1 Specific interest of ATD items

The model of accumulation of ATD with respect to the crisis point does not separate the ATD from its interest. It was not possible to do such separation from the data analyzed during the current investigation. In order to further develop the models shown here, it's of utmost important to understand the interest associated with specific ATD items, in order to prioritize them and to understand which one contributes more to the total accumulation of extra-effort leading to a crisis point. An initial step has been done by the same authors of this paper in a recent paper accepted for publication [42].

7.8.5.2 Other socio-technical patterns related to ATD accumulation and refactoring

As we have highlighted in the case study, ATD is not only a technical problem, but involves several factors, from the psychological one to the organizational structure, to the processes. Understanding socio-technical patterns that favor (or limit) how ATD is accumulated might be useful for the proactive avoidance of its accumulation.

7.8.5.3 Further evaluation

The study of the ATD is quite recent and needs further evidences. We have provided multiple sources of evidences for some results, but for other, of more exploratory nature, the scientific community needs further evaluation. We have compiled Table 22, which shows the results that need more evidences. Such results should not be considered unreliable, since they are based on the combined experiences of several employees from 7 sites in 5 large software companies. However, more in depth studies

might be done in order to define more precise models and compare with multiple sources of evidence. We have done a first step in such direction.

7.8.6 Threats to validity

We discuss the threats to validity with respect to the guidelines proposed in [63]: construct, internal, external validity and reliability.

7.8.6.1 Construct validity

We did not use the ATD terminology (*debt*, *principal*, *interest*) during the investigation, since it could have been interpreted differently in different contexts. In order to keep construct validity, we operationalized the ATD as "architecture inconsistencies" (or alternatively "sub-optimal solutions" or in some cases "violations", depending on the class of ATD) with respect to the current desired architecture related to a specific case. With the same approach we operationalized the *principal* as the refactoring cost to obtain an optimal solution and the *interest* as the extra-effort caused by the ATD or other kinds of extra-impact. We investigated the actual existing cases with software and system architects, obtaining the best knoweledge about both the desired architectures and a good explanation of the sub-optimal solutions in place. As for the principal and interest, we interviewed the developers involved in the case studied, in order to be sure that estimations and evaluation of effort were as accurate as possible.

7.8.6.2 Internal validity

Rather than investigating a direct cause-effect relationship, we have collected *arcthitecture explanations* [66] from several cases, in order to understand which factors caused the accumulation of ATD and how such accumulation is handled in the specific cases. Since we aimed at understanding the causes of ATD accumulation, there exist a threat to internal validity. We need to take in consideration the possibility that other factors than the ones listed here would cause ATD, and that crises would be reached because of other projects' issues. We mitigated this threat by obtaining the information about the causes of the same ATD item from several roles, and we asked follow-up questions in order to probe the explanations. Also, collecting similar evidences supporting the same explanation across the cases contribute to strenghten our conclusions.

7.8.6.3 External validity

One of the major threats for case-study research is the ability to generalize from the case-specific results to other cases. Generalizing to a universal theory is not necessarily the goal for an engineering discipline: according to [66], middle-range theories, valid to a restricted ranges of contexts, result more useful in practice. In order to develop such middle-range theories, we have employed an analytical induction strategy to generalize from case-studies [66]. We have collected architectural explanations from contexts that are architecturally similar among themselves, but contain some differences (all of them are large companies developing embedded software and having similar organizations). The similaries allow the researchers to make the claims more robust (for example, the same code was mentioned in three of the 7 cases), while the differences allow the extension of the findings, if similar, to such contexts as well. In our case, we have identified the findings that were confirmed by multiple sources (for example, the model concerning the crisis point was found repeatedly valid from all the sources) and the ones that need further investigation, in Table 22. As for the second kind of results, we don't say that such findings are not valid in all the contexts, but rather that it was not possible, with our data, to find strong confirmatory results. The table can be used to understand how much our results can be generalized based on the kinds of evidences

obtained from the different sources. Although we don't claim the results universally general, we can say that for RQ1 and RQ2 the findings are supported by confirmatory findings crossing multiple contexts, and therefore can be considered reaching a generalizability of middle-range theories, where the range is represented by large Scandinavian software companies developing embedded software and employing ASD. As for RQ3, we report propositions that need to be further evaluated with different means than the ones employed in this study.

7.8.6.4 Reliability

As described in the methodology section, three authors participated in defining the research design, the questions and into checking the findings from the coding activities. The results were presented in a workshop at the end of each of the four phases to the industrial contacts involved in the investigation, in order to collect feedback and strenghten the reliability of the results.

7.8.7 Related Work

Lehman et al. [122] 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 (Figure 32), 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. [39]. 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 [35]. Sindhgatta et al. [123] 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.

7.9 CONCLUSIONS

Decisions on short-term and long-term prioritization of architecture refactoring need to be balanced and need to rely on the knowledge of the underlying phenomenon of ATD. The current management of ATD is an under-researched topic and we contribute to the empirical software engineering body of knowledge by reporting from a multiple case study investigating practitioners' experiences from 7 large Scandinavian companies employing Agile and developing product lines of embedded software.

In this paper 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 (RQ1), therefore easing the ATD management for architects and managers.

We have also presented 2 models for describing the accumulation and refactoring of ATD over time (RQ2). Such models are the *Crisis Model* and the *Phases Model.* Such models can be further studied and tested with the conduction of experiments and the collection of quantitative data by the ISERN community.

Based on the models, we have identified possible strategies for refactoring ATD (RQ3) and we provided recommendations with respect to the minimization of development crises. We conclude that *complete refactoring* is not a possible strategy in

the current studied companies, due to the continuous and inevitable accumulation of ATD and the impossibility of removing it all. The *No refactoring* strategy leads to crises points often, hindering the long-term responsiveness in providing new customer value, as required in ASD. The best strategy is therefore to apply *partial refactoring* to minimize crises and to push the crisis point as far as possible with respect to the lifecycle of the products. The results highlight different outcomes related to different ATD prioritization strategies, which would help architects and managers in balancing the ATD management strategies with respect to the business goals and the life-cycle of the products.

An important goal in research and industry is to improve the practices and tools to uncover ATD present in the system and to keep track of it. 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.

8 ARCHITECTURE TECHNICAL DEBT PHENOMENA HINDERING LONG-TERM RESPONSIVENESS

In this Chapter we have investigated which Technical Debt items generate more effort and how this effort is manifested, hindering short-term or long-term responsiveness, during software development. We conducted a multiple-case embedded case study comprehending 8 sites at 6 large international software companies. We found that some Technical Debt items are contagious, causing other parts of the system to be contaminated, which leads to the growth of interest. In order to monitor the increasing interest we have investigated 12 concrete cases of Architectural Technical Debt, finding important factors to be monitored to refactor the debt before it becomes too difficult. We also identify a second socio-technical phenomenon, for which a combination of weak awareness of debt, time pressure and refactoring creates Vicious Circles of events during the development. Instances of these 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.

This chapter has been submitted for publication as:

Martini A., Bosch J.: *"Contagious Technical Debt and Vicious Circles: a Multiple Case-Study to Understand and Manage Increasing Interest"* submitted to

A short version of this chapter has been published as:

A. Martini and J. Bosch, "The Danger of Architectural Technical Debt: Contagious Debt and Vicious Circles," in *2015 12th Working IEEE/IFIP Conference on Software Architecture (WICSA)* [42].

8.1 INTRODUCTION

Large software industries strive to make their development processes fast and more responsive, minimizing the time between the identification of a customer need and the delivery of a solution. The trend in the last decade has been the employment of Agile Software Development (ASD) [6]. 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 [38]. Tom et al. [33] 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)[38]. 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 [124].

ATD has been recognized as part of TD, but the various ATD items 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 high cost of architectural changes, a challenge for software companies is to prioritize the refactorings 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 in terms of effort paid later (interest)?

RQ2: What are the effects, (interest) triggered by such ATD items?

RQ3: What are the socio-technical anti-patterns that cause the ATD interest to increase over time?

RQ4: How can the increasing interest be identified and stopped?

The main contributions of the paper are:

- We have qualitatively developed and validated (through multiple sources) a taxonomy of effortful ATD 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.
- To inform RQ4, we have identified two main kinds of interest, the interest on the principal and the interest on other factors. Such factors lead to the increment of interest of an ATD item. By monitoring the growth of such factors, we show how the companies can proactively avoid the increment of interest.

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

8.2 Architecture and Technical Debt

8.2.1 Definition of ATD

ATD is regarded [33] as "sub-optimal solutions" with respect to an optimal architecture for supporting the business goals of the organization. Specifically, we refer to the architecture identified by the software and system architects as the optimal trade-off when considering the concerns collected from the different stakeholders (which is usually a "desired" architecture). In the rest of the paper, we call the sub-optimal solutions *inconsistencies* between the implementation and the architecture, or *violations*, when the optimal architecture is precisely expressed by rules (for example for dependencies among specific components). However, it's important to notice that (in our studied cases) such optimal trade-off might change over time, as explained in this paper, due to business evolution and to information collected from implementation details. Therefore, it's not realistic to assume that the sub-optimal solutions can all be identified and managed from the beginning. For this reason, it becomes important to continuously monitor architecture and ATD rather than relying only on the upfront design.

8.2.2 Previous research on ATD

The term *Technical Debt* (TD) has been first coined at OOPSLA by W. Cunningham [32] to describe a situation in which developers take decisions that bring short-term benefits but cause long-term detriment of the software. The term has recently been further studied and elaborated in research: in 2013 Tom et al. [33] conducted an exploratory case study technique that involves multi-vocal literature review, supplemented by interviews, in order to draw a first categorization of TD and the principal causes and effects. In such paper we can find the first mentioning of Architectural Technical Debt (ATD, categorized together with Design Debt). A further classification can be found in Kruchten et al. [38], where ATD is regarded as the most challenging TD to be uncovered since there is a lack of research and tool support in practice. Finally, ATD has been further recognized in a recent systematic mapping [31] on TD. Such recent research highlights the gap in the current scientific knowledge, which gives us the motivation for this work.

8.2.3 Previous research on management of TD

Some studies have been conducted on the management of TD, also supported by a dedicated workshop (MTD), usually co-located with premium conferences, such as ICSE and ICSME.

A first roadmap has been created in 2010 by Brown et al. [34]. In 2011 Guo et al. proposed an initial portfolio approach with the creation of TD *items*. The same authors proposed a further empirical study on tracking TD [35] Seaman et al. identified the theoretical importance of TD as risk assessment tool in decision making [36]. TD has also been used for defining part of a method for assessing software quality, SQALE [37]. Such model has been also implemented in a tool, but the main support is currently given on a source code level (very limited on the ATD aspect).

8.2.4 Models for technical debt

The studies in TD are quite recent, and the subject is not mature. Some models, empirical [39] or theoretical [40] have been proposed in order to map the metaphor to concrete entities in software development. We use, in this paper, a conceptual model comprehending the main components of TD:

8.2.4.1 Debt

The debt is regarded as the actual technical issue. Related to the ATD in particular, we consider the ATD item as a specific instance of the implementation which is sub-optimal with respect to the intended architecture to fulfill the business goals. For example, a possible ATD item is a dependency between components that is not allowed by the architectural description or principles defined by the architects. Such dependency might be considered sub-optimal with respect to the *modularity* quality attribute [41], which in turn might be important for the business when a component needs to be replaced in order to allow the development of new features.

8.2.4.2 Principle

It's considered the cost for refactoring the specific TD item. In the example case explained before, in which an architectural dependency violation is present in the implementation, the principle is the cost for reworking the source code in order to have the dependency removed and the components not being dependent from each other.

8.2.4.3 Interest

A sub-optimal architectural solution (ATD) causes effects, which have an impact on the system, on the development process or even on the customer. For example, having

a large number of dependencies between a large amount of components might lead to a big testing effort (which might represent only a part of the whole interest in this case) due to the spread of changes. Such effect might be paid when the features delivered are delayed because of the extra time involved during continuous integration. The most important findings in this paper are related to this component of Technical Debt.

8.2.5 The time perspective

TD is strongly related to time. Contrarily to having absolute quality models, the TD theoretical framework instantiates a relationship between the cost and the impact of a single sub-optimal decision. In particular, the metaphor stresses the short-term gain given by a sub-optimal solution against the long-term one considered optimal. Time wise, the TD metaphor is considered useful for estimating if a technical solution is actually sub-optimal or might be optimal from the business point of view. Such risk management practice is also very important in the everyday work of software architects, as mentioned in Kructhen [43] and Martini et al. [44]. Although research has been done on how to take decision on architecture development (such as ATAM, ALMA, etc. [45]), there is no empirical research about how sub-optimal architectural solutions (ATD) are accumulated over time and how they can be continuously managed.

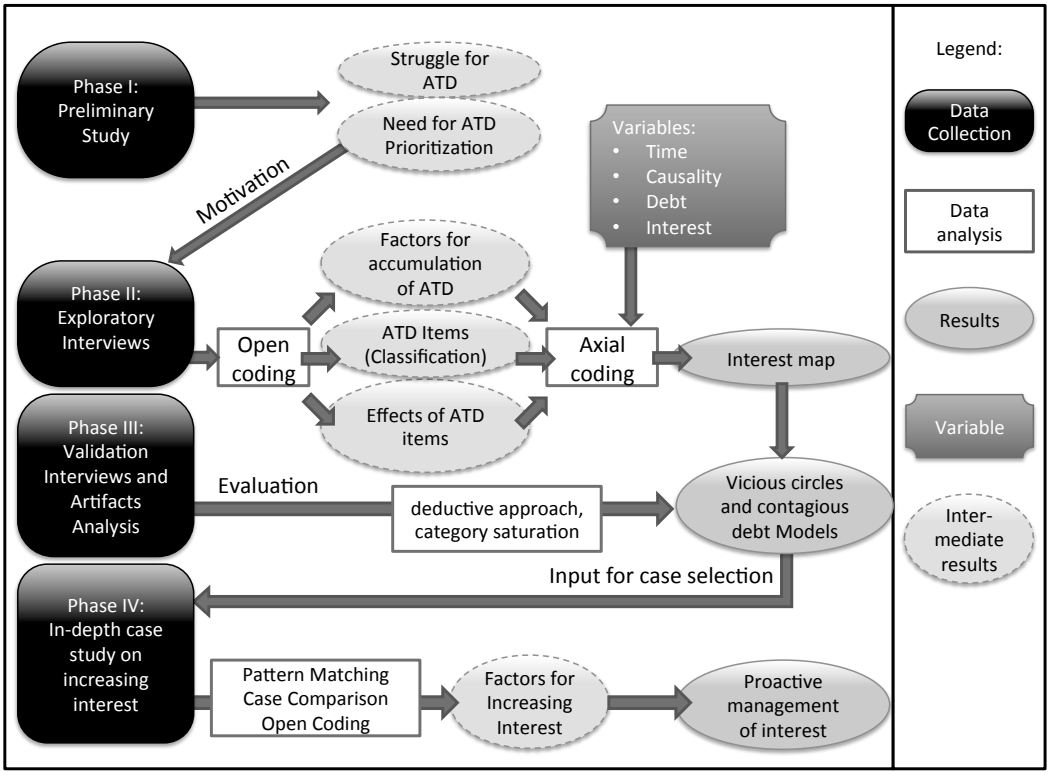


Figure 42. Our Research Design: data collection, inductive and deductive analysis and results based on different sources for triangulation

8.3 RESEARCH DESIGN

We conducted a 2-year long, multiple-case, embedded case study involving 8 Scandinavian sites in 6 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 [63] (collecting supporting evidence from different sources rather than from a single context). The rationale for such a longitudinal investigation (including several case studies rather than reporting a single snapshot of one organization), is explained in [77] as *systematic combining*: since the goal of

research is to match the theory with the empirical world, every time we look at the empirical evidence we might discover something new that needs to update the theoretical framework and the previous theories. This iterative and continuous approach is in line with the need of flexibility in case studies suggested in [63]. By replicating cases and adding new information coming from similar but not identical contexts, we could discover new variables and factors that were important both for theory building (for example the contagious debt model), but also for the evolution of the framework itself (in this paper, we update the theoretical framework of TD by redefining the interest to more factors rather than only one). The nature of the study was mainly 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. On the other hand, in the later phases (I-IV, see 8.3.2), we aimed at collecting more evidences that would strengthen the theoretical power of the previously defined theories (for example, in phase IV we collected an additional 12 cases in which we could match the *contagious debt* pattern). The research design is outlined in Figure 42.

8.3.1 Case Selection

We have employed an embedded multiple-case study [63], 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 9. We selected, following a literal replication approach [62], 4 companies: A, B, C (3 sub-cases) and D, F 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 [62] (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 4 different departments within company C (C_1, C_2, C_3, C_4). *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. *Company F* is developing devices used for defense systems. Some of the contextual factors relevant for the investigation are visible in Table 24.

8.3.2 Data collection and analysis

We planned a 4-phase investigation of the ATD items and effects (described in Table 23). The phases (black boxes) and their results are visible in Figure 42. The properties and the number of participants are also summarized in Table 1. For each phase we describe the data collection procedure and the analysis that followed. All the interviews were recorded and transcribed. The analysis was done following approaches based on Grounded Theory [55] and from case-study research [62][63], frequently employed for the analysis of large amount of semi-structured, qualitative data representing complex combinations of technical and social factors. For the analysis we used a tool for qualitative analysis, (atlas.ti), which allows the categorization and analysis of qualitative data according to the approaches described in [55] and keeps track of the links between the codes and the quotations they were grounded to, in order to create a *chain of evidence* [63][62].

Table 23 Properties and numbers related to data collection

Phases of data collection	N.participants	N.sessions	Companies	Roles involved
Phase I (Preliminary interviews)	25	3	Company-specific	Developers, architects, testers, line managers, Scrum m.
Phase I (Evaluation workshop)	40	1	Cross-company	Developers, architects, line managers
Phase II (group interviews)	26	7	Company-specific	Developers, architects, product owners
Phase II (evaluation workshop)	10	2	Cross-company	Architects, line managers
Phase III (evaluation interviews 1)	10	1	Cross-company	Architects, product owners
Phase III (evaluation interviews 2)	12	1	Cross-company	Architects, developers, scrum masters
Phase III (validation workshop)	20	2	Cross-company	Architects
Phase IV (increasing interest interviews)	39	8	Company-specific	Architects, developers, (other stakeholders involved)
Informal interaction	7	NA	Company-specific	Software and system architects

8.3.2.1 Phase I – Preliminary study

Data Collection - We conducted 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. This phase contributed in identifying and selecting what Research Questions (RQ1-3) were critical to be answered for the studied industrial setting. This way, we could complement the literature review, assuring that answering the RQs was not only important for a theoretical point of view, also for a practical need. 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 validated and discussed in a final one-day workshop involving 40 representatives from all the 8 cases (see Table 1).

In the preliminary study we asked open questions:

- "How do you control consistency between the implementation and the architecture?"
- "Which architecture risk management activities are you employing on different level of abstraction?"
- "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.

Data Analysis - The data were analyzed in an explorative manner, using a technique called "open coding" analysis, suitable for exploratory: We then filtered the codes into "challenges". Then, by the inductive categorization of the codes, we could see how several statements, consistently throughout all the cases, fell in the following categories:

- "reactive behavior to architecture drifting" (abbr. RBAD)

- "lack of continuous risk management activities for architecture drifting" (abbr. LCRMAD)
- "down-prioritization of architecture" (abbr. DPA).

The combination of these three categories leads to the phenomenon of accumulation of Architecture Technical Debt (ATD): sub-optimal solutions, the debt, are not evaluated continuously (RBAD), the risky effects are not understood in time (LCRMAD) and therefore the refactoring is down-prioritized.

The preliminary study showed 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 42), which led to the next phase.

8.3.2.2 Phase II – Investigating ATD classes of items and their effects (interest)

Data Collection - 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. Such interviews were aimed at answering the first 3 Research Questions (RQ1-3). 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 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.

Data Analysis - First we analyzed the data in search for emergent concepts following the *Open Coding* technique [55], which would bring novel insights on the analyzed issue. We coded the identified ATD items in a taxonomy (*ATD Items Classification* in Figure 42). We used the same technique for identifying the key effect phenomena (*Effects of ATD items* in Figure 42) and the causing factors that were related to each item (*Factors for Accumulation of ATD* in Figure 42).

We then apply the *Axial Coding* approach [55]: the codes and categories were compared in order to highlight connections orthogonal to the previous developed categories. Such analysis showed which category of items (in the *ATD Items*

Classification, Figure 42) was connected to which effects (*Effects of ATD items* in Figure 42). This way we could build the complete *Interest Map* visible in Figure 42, in which we could represent the *debt* and the *interest* discovered through our investigation.

8.3.2.3 Phase III – Evaluation interviews and Artifact Analysis

Data Collection - 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 8.5.1) 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.

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 provide cases to validate the models.

Data Analysis - We applied the *pattern matching* [62] deductive approach for evaluating the models of *Contagious Debt* and the *Vicious Circles:* the technique consists of using existing theoretically developed models (developed in Phase II) in order to match empirically provided patterns.

8.3.2.4 Phase IV – In-depth study of increasing interest

The models of *Vicious Circles* and *Contagious Debt* suggested that for some ATD items the interest was continuously increasing, leading to severe development crises ([125]). We therefore decided to follow-up with another multiple case–study. We chose to conduct a multiple embedded case-study [63], [62] where the unit of analysis was the ATD item studied. The company, where the ATD item was studied, was considered in order to analyze *context factors* for the increment of interest (see Table 24). For example, for company B we identified the intense interaction with an Open Source community as a factor for the (non-) increment of interest. The cases were also useful for further evaluation of the Contagious Debt and vicious circle models.

Data collection - The investigation was structured in different steps (for which we developed and maintained an interview protocol, as recommended in [63]).

Preliminary Workshop - We performed, at each studied site, an initial investigation with the following purposes of identifying suitable cases with known or suspected high increment of interest. We carried out this activity with the architects and we used the model developed in Phase II and evaluated in Phase III in order to elicit suitable cases that seemed to be contagious (see arrow *Input for case selection* in Figure 42). This part also helped minimizing possible construct validity threats, by aligning the concepts and the characteristics of the cases among the companies. We selected cases that were similar because they reported of growing interest, but also seemed to have different *context factors*, such as different stakeholder involved or to belong to different ATD classes. This way, we tried to apply a replication strategy [62], having similar cases from the point of view of the main phenomenon studied, but that would slightly differ from each others, in order to maximize the variance. Such strategy allowed to find more factors for the same phenomenon, dependent or not on the context. The

preliminary investigation lasted between 30 minutes and 1 hour and also helped the identification of participants: we prepared the real investigation by selecting the suitable people able to provide the important information: the developers involved during the development, in order to understand how the ATD item was injected in the system and how it became contagious, and the developers, testers and other possible stakeholders involved in the effortful of evolution and maintenance of the studied ATD item.

Investigation workshop - In the investigation workshop, we first asked for an high level explanation of the issue: the participants provided a description of the ATD item(s) relating to the context. A short description of such issue is reported in Table 24. Then, we performed a structured interview focused on the cases and following the script:

- $\mathbf{T_{now}}$ analysis. We performed an analysis of the current situation for the ATD item: we investigated the *Source* of the problem, the *Current Principal* and the *Current Interest.*
- $\mathbf{T_{past}}$ analysis. We asked how the ATD was *injected* in the system and how that happened. We then focused on the *Propagation of the ATD*. It was important to understand at what point the interest of the ATD started to grow. By identifying the point in time and the factors that led the ATD to be propagated, we aimed at identifying what to monitor in order to avoid the contagion (*Factors for Increasing Interest* in Figure 42). We asked this question only if we understood that the ATD item was already propagated. In some cases, the propagation was still hypothetical, so we did not need to analyze previous propagation, but we focused on the *future propagated interest* (explained below in $\mathbf{T_{future}}$ analysis). In other cases, the ATD item was refactored in time because of the estimated growth.
- $\mathbf{T_{future}}$ analysis. We asked the participants to estimate the growth of the impact of the ATD item and the growth of the refactoring, in order to understand what was their current perception of the *Factors for Increasing Interest* and if it would be possible to use the current estimation for the growth in order to decide for the refactoring.

8.3.2.5 Data analysis

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

- Description of the item
- Interest (or impact, current or predicted)
- Principal (or cost of refactoring, current or predicted)
- Factors for Increment of Interest
- Contextual factors

We then applied the Explanation Building analysis strategy defined in [62]: the purpose is to define causal links by using a narrative approach (used by most existing case studies [62]). The results from such analysis are shown in Table 24, followed by the presentation of the *Factors for Increment of Interest*. The newly found explanations were then analyzed in order to provide guidelines for *Proactive Management of Interest*.

In this phase we also used the *pattern matching* analytical strategy recommended in [62]. Such strategy is used to verify the existence of previously formulated patterns

(hypotheses) after a new collection of data. In our cases, we used the models obtained by the data collection conducted during phase II and III as *pattern tests* to be used during phase IV to match the pattern. This kind of evidence contributed in evaluating the formulated models related to *contagious debt* and other *vicious circle*.

8.4 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 [46],[126]. 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 43) 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, which can be considered a major threat for software development. We will then present the results about how the interest increases, showing the major factors for its increment and then what guidelines can be applied in practice to proactively avoid it.

8.4.1 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 43.

8.4.1.1 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 cases 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 43). 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 8.5.1.

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).

8.4.1.2 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 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 phenomenon involved specifically the feature teams interviewed at company C. In such context, the teams were 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 8.5.

8.4.1.3 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 it is 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.

8.4.1.4 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 8.5.2).

8.4.1.5 Unidentified non-functional requirement (NFRs)

Some NFRs, 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.

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.

8.5 VICIOUS CIRCLES

We have previously 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 [125]: they are *vicious circles*.

Such vicious circles are well visible in our model in Figure 43: the column on the left includes the possible causes of ATD accumulation ([125] for details), which we have represented with black boxes (*Cause of ATD generation*). Among the phenomena

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.

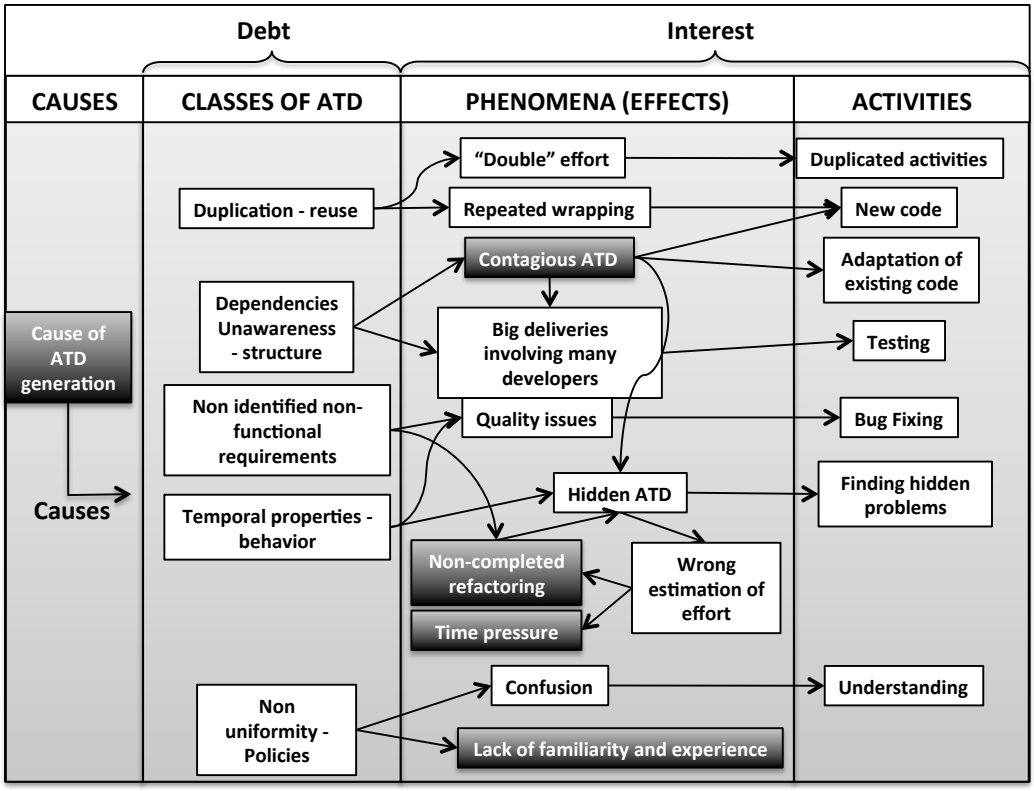


Figure 43. The model shows the causes for ATD accumulation (black boxes), the classes of ATD (which represent the Debt), the phenomena caused by the items and the final activities (which together represent the interest to be paid).

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 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. It's important to notice that the worse case might not be consisting of non-linearity: if the linear accumulation is steep enough, the crisis might happen earlier than in the non-linear case.

The highlighted vicious circles are:

8.5.1 Contagious ATD

We have introduced the concept of contagious ATD in section 8.4.1.1. Contagious debt can be defined as *an ATD item whose source affects other parts of the system over time, causing the sub-optimality (the debt) and its interest to grow*. Some examples are reported in Table 24, and a more detailed example is explained in section 8.6.1.1. We therefore point the reader at those references for a deeper understanding of the phenomenon.

8.5.2 *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 [125] and [127] (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.

8.5.3 *Propagation by bad example*

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.

This phenomenon involved specifically the feature teams interviewed at company C (see also the case Case_C1_2 in Table 24). In such context, the teams were 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. It's important to notice that such pattern can be repeated over and over again. In the studied case, the same violation of the non-allowed dependency rules brought to have a huge number of non-allowed dependencies in place, which represent a high lack of modularity in the system, for which a lot of interest is paid.

8.6 UNDERSTANDING AND MANAGING THE INCREMENT OF THE INTEREST

In section 8.4.1 we have presented a map of the ATD classes and their interest. In section 8.5 we have identified which vicious circles lead to continuous accumulation of interest. In this section, we present the results that would explain the factors involved in the increment of the interest and how the increment of the interest can be proactively avoided or managed. In order to do this, we first present the collected cases, to help the reader understanding the contexts. Then, we present the findings related to the increment of interest: the two main kinds of interests (interest on the principal and interest on other factors), then we explain why their differentiation is important and how to manage them in different ways with respect to an optimal refactoring strategy.

8.6.1 Presentation of the cases

First of all, we report all the cases specifically recorded for studying contagious debt. We highlight the key attributes for the cases, for example the description, the contagiousness and the contextual factors that might be useful for understanding the context of the problem.

Table 24. Cases of ATD analyzed with increasing interest

Case Id	Comp	Status	Description	Growth of interest	Contextual factor
Case_B_1	B	Refactored	Lack of good "communicatio n mechanism" for different applications sharing a memory resource. Such TD created quality issues (bugs) for the users (customers)	Several new (external) applications were going to use the shared resource. The interest affected the testware: with the refactoring, it was estimated to have been saved 20% of the test time. The refactoring was not growing anymore, many quality issues did not require fixing or workarounds. Most of the time was spent in refactoring the application, which shows how much interest it was accumulated already.	Open source software community (external) pressure to remove TD
Case_B_2	B	Refactored	Quick fix of data structure to allow performance for a single project.	If the solution was rolled out for all the other projects, it would have had a lot of ripple effects, since the data structure was used in several place.	Open source software community (external) pressure to avoid the TD
Case_B_3	B	Propagated	The old version of an external library was used because it was difficult to adopt the new one.	The effort required to change the library has grown since the first release of the new library. Several updates now have to be refactored in order to introduce the new library	Open source software community (external) pressure to remove TD
Case_$C1_1$	C1	Partly Refactored	A common component was not designed	The refactoring was growing at least linearly with the number of application added. Once refactored, the growing	Reuse is not planned. Other applications

			optimally. Some applications have started using it. The component has been refactored, but the applications using it cannot be refactored because of other priorities, while new applications are using the new version.	stopped, but the double maintenance (for the two versions of components) remained. The number of external users was also growing.	requested to use the component.
Case_C1$_2$	C1	Propagated	There is a non-allowed dependency between two components.	Every time a new feature is added, the dependency is made bigger and bigger, meaning that new information has to be known about the status of a component by the other. Refactoring is too costly to be put as story for the dependency	The team was not familiar with the code, so they understood the dependency late and they did not know that there was TD.
Case_C3$_1$	C3	Injecting	A component is being reused and adapted in order to save time for the short-term delivery of a customer feature that would allow the customization of some functionalities for the user. Instead, a better version is planned to be coded, which would bring better NFRs.	There is a steep, linear, increasing interest of the refactoring cost with respect to new features added to the system and an increasing impact on several NFRs.	Legacy: existing of previous system made of a collection of different architectures
Case_C3$_2$	C3	Injected	A mediation layer is written without satisfying scalability requirements.	Several new applications are going to be affected by the scalability problem. Also, complexity would grow because of the sub-optimal mediation layer, and it would	

			The solution is going to work now, but cold be refactored to be scalable for later.	be exposed to the new applications.	
Case_D_1	D	Propagated and propagating	An interface is growing sub-optimal because of the backward compatibility required by previous customers.	For every new product the cost of refactoring grows linearly. The interest is also at least linear with respect to the number of new products, and consists of extra-testing, higher complexity and defect proneness.	Long-living products, backward compatibility requirement, low level embedded software developed.
Case_D_2	D	Propagated and propagating	An internal interface is not well defined. A large number of components are already using it, which will need to be refactored together with the interface.	For every new component developed, the cost of refactoring grows linearly. The interest is also at least linear with respect to the number of new components. New people have been hired and the TD is going to be propagated to their knowledge (they will have to be re-trained with the new system). Since the complexity is also growing, the time to refactor will increase.	Long-living products, backward compatibility requirement, low level embedded software developed.
Case_E_1	E	Injected	A database component does not provide a standard API, and an application is using the private API.	Several new applications are going to be developed and connected with the database component. If the standard API is not in place the new application would access the private API and it would become very costly to evolve the database component afterwards, since it would require to change all the accessing applications.	Non-embedded system.
Case_F_1	F	Propagated	Business logic was embedded in the dialogs of the UI.	Every time a new dialog is added to the system, the debt is propagated.	Team investigation, design TD

8.6.1.1 Increment of interest

In this section we highlight the key findings derived from the previous cases. The first important property is that an ATD item changes with the change of the system around it. *An ATD item that seems not to be contagious today, might become contagious tomorrow*. Therefore it becomes important to monitor the known ATD.

An important distinction also emerges from the cases. The interest is composed by two elements:

- the growing cost of refactoring (interest on the principal, I_P) and
- the growing impact (interest, I).

Both these elements can be considered interest, since the principal, according to the financial metaphor, is to be considered only the original cost of repaying the source of the original ATD item: for example, the cost of refactoring an APIs, but not the components using it. We describe them in the following sections, by understanding how the contagiousness depends on different factors. In the end, we highlight how the various factors interplay together and how the practitioners can monitor the factors in order to prioritize the refactoring at the right point in time.

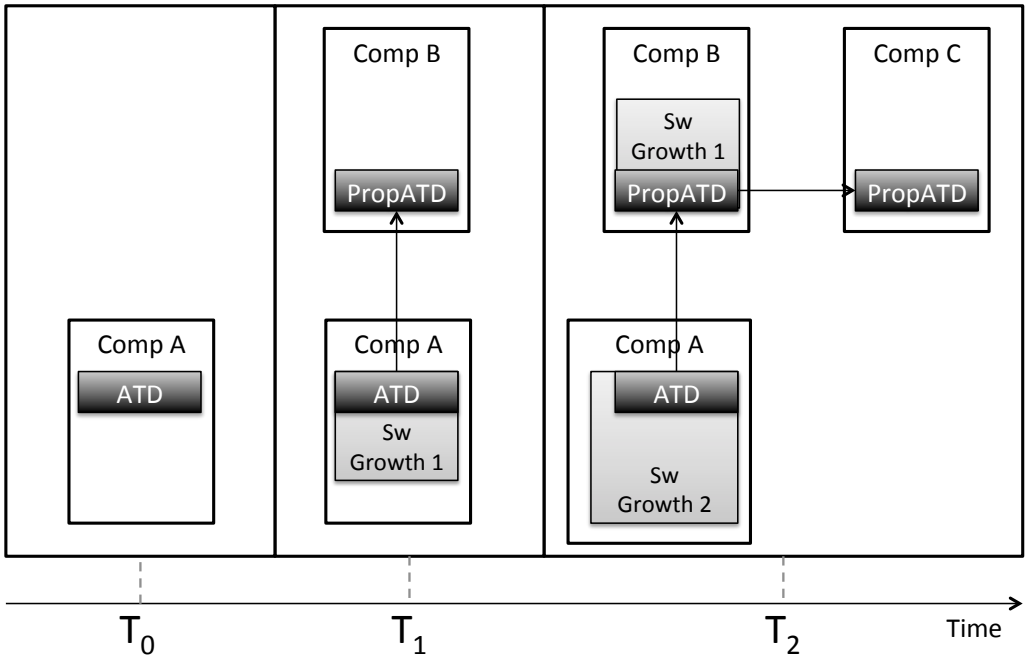


Figure 44. Model for Contagious Debt accumulation at different points in time: T_1, T_2 and T_3

8.6.1.2 Increasing cost of refactoring (interest on the principal, I_p)

Although in the TD metaphor the interest would be separated from the principal, it's clear from the cases that the interest involves the cost of refactoring as well. In all of the cases, the cost of refactoring was raising, at least linearly with respect to the added software over time. As we can see in one of the refactored items (see Case_B_1), the developers said that most of the refactoring was not concerning the initial principal (the source of the TD), but was concerning the applications built on top of it.

We will describe a model of the anti-pattern concerning the contagious debt for what concern the cost of refactoring, that we call Interest on the Principal (or I_p).

In this case [Case_E_1], 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 every time 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 Figure 44: 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 ($Case\_E_1$): 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 components grow constantly, making it a worse case scenario, but we have to consider the fact that also more components could have been added.

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

8.6.1.3 Factors related to the increasing of Interest (I_f)

From the cases studied, it's clear that *the interest (impact) of an ATD item is not only to be related to maintenance, but it has ramification on other factors*. It's therefore

important to understand what are these other factors in order to monitor their growth (together with the growth of the system described before). We highlight the main factors below. One of the main implications is that the interest is not only a property of the source code, but it includes other artifacts and other parts of the organization.

Complexity – Although the complexity can be regarded itself as TD, in this case we take the perspective of it being a factor for the growth of the interest related to another ATD item. As clear from several cases, the growth of complexity makes both maintenance difficult (growing of I) and the refactoring costly (growing of I_p). The implication is that the growth of complexity needs to be monitored in those parts of the software that are connected with an ATD item.

Testware – In many cases analyzed, the interest is related to the testware: often more tests need to be in place or resources for testing (for example, time). Another important point is that refactoring ATD implies, often, refactoring the related tstware as well. Monitoring the growth of the testware in relationship to code affected by ATD is therefore important to understand the if the interest is increasing over time.

Users – A component might be used by a number of other internal components, applications (or features) or internal or even external systems. For example, in several cases the increment of interest was considered linear to the number of features included in the roadmap. In those cases where the interface with ATD was exposed, the teams developing something related to such interface were affected by the interest related to the ATD. By understanding how many users are connected to the ATD, it's possible to decide to refactor an item before the number of users, experiencing the inconvenient effects related to the interest, would grow, minimizing the costs. Another reason why the number of users is important is clear in case D_2, where the training of the newcomers would be more effective if happening after the refactoring: this way, the time for learning is limited to the new system and not to the old system as well. Consequently, monitoring the number of users might be related to information coming from the Human Resources. In the cases related to company B and C2, the users where external to the organizational boundaries, i.e. belonging to the ecosystem. In such case, the interface would be used by an undefined number of users: such information constitutes a risk factor that needs to be taken in consideration. In all the cases including external users, it was decided to refactor the ATD, even if in $Case\_C2_1$ the refactoring was not completed, causing an internal extra-cost, but allowing the external users to not experiencing the interest of ATD.

Non-Functional Requirement – in many of the cases, the problems related to interest where not only coming from maintainability, but where involving other NFR and where estimated to worsen NFR for the new part of the system developed. It's therefore important to monitor how ATD is going to affect the new system with respect to the NFR that are important to the company.

8.6.2 Monitoring the growth of the factors (I_f) in order to optimize the benefits of refactoring

The usual rationale for refactoring ATD is the evaluation of the cost of the whole refactoring (that can be considered as the cost for refactoring the source of ATD plus the cost for refactoring the increment of principal, I_p) versus the remaining cost of the interest (I_f). According to the cases, when the ATD item was discovered, the cost and probability of I_f and I_p was low, and therefore the ATD was found not convenient to be refactored according to the participants. However, when the interest I_f became too much and the participants wanted to refactored the ATD item, the growth of I_p had already happened, causing a struggle in deciding for refactoring because of the increased total cost and the cost already spent on the interest I_f. It seems clear that *the more the interest on the principal (I_p) grows, the less convenient and likely it becomes to remove the interest I_f.* Also, it's to be considered that in practice, large tasks are

difficult to be approved by the management and to be adjusted together with the feature development.

Another important consideration is that the growth of I_p and I_f are not necessarily known. According to the previous results, obtained from the cases, both I_p and I_f increase with the growth of the system and the growth of the factors (by definition of the two interest indicators). What happens in practice is that the growth of the factors is not necessarily known long in advance (or is considered not probable), but it appears when the company starts paying I_f. However, at such point the factors are already grown and therefore I_p is grown (which, as stated before, prevent the management to prioritize the refactoring). Therefore, rather than monitoring the cost spent on the interest based on the factors (I_f) in a reactive manner, it appears to be better to estimate in advance the growth of the factors themselves: for example, using the roadmap it's possible to know how many new features are going to be developed, and which part of the system will most probably grow, which in turn might anticipate the growth of I_p and avoiding reaching the point when refactoring the ATD is too costly. In conclusion, by monitoring the growth of the factors, the company would better know when and if to refactor the ATD with as low I_p to be paid as possible, but at the same time avoiding as much cost of I_f as possible, which in the end would be the optimal choice (visible in Figure 45).

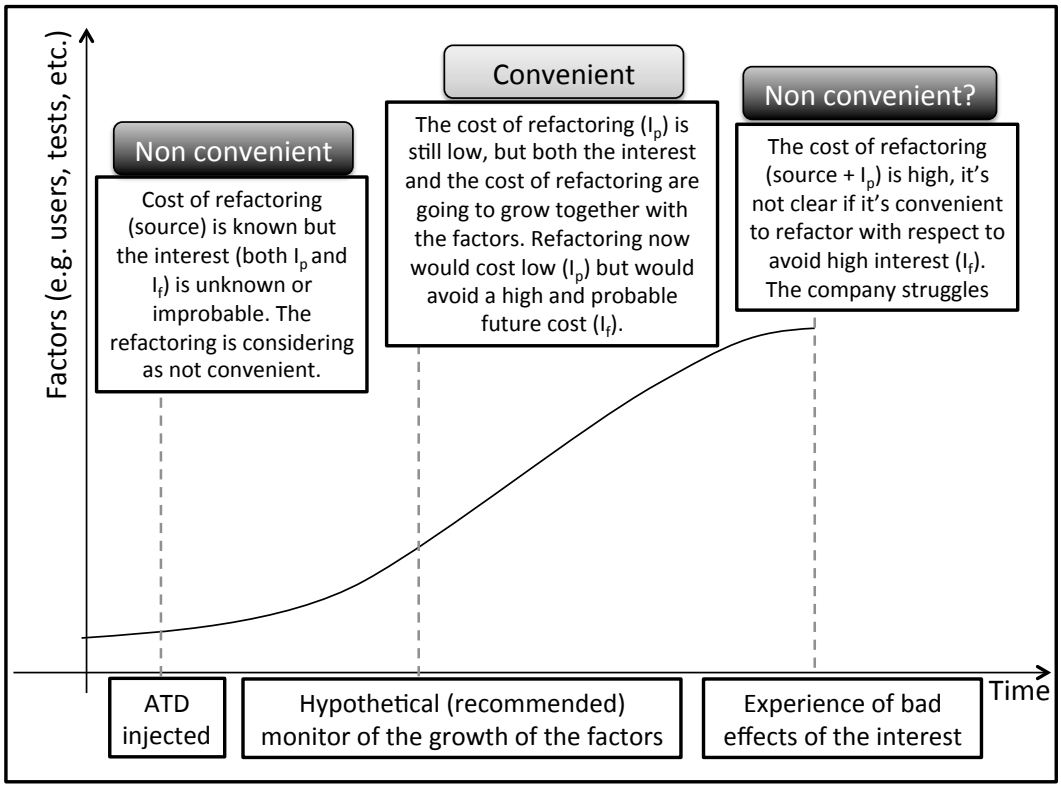


Figure 45 By monitoring the factors, it's possible to pay low to refactor growing interest

8.7 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 8 organizations. To inform RQ2 we have built a model of the effects (Figure 43) 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. As for RQ4, we have investigated 12 cases of contagious debt, in order to understand which factors need to be monitored to avoid the contagiousness and therefore the payment of high interest.

8.7.1 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.

Enabling iterative and proactive ATD management: 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. The dangerous classes, interest activities, vicious circles, and the factors for the increment of the interest that we propose here can be useful to support iterative architectural retrospective. 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. The cases show how some ATD items belonging to some classes (for example the contagious one) usually creeps in the system because its interest is not evaluated continuously. When finally the interest becomes visible as a software development crisis, it's usually too late to refactor the ATD item. In order to recognize this phenomenon, we have found, through the analysis of 12 cases, the factors that can be monitored in order to identify the increment of interest, which would facilitate the refactoring strategy.

Prioritization of ATD items based on the magnitude of the interest: there might be a lot of suboptimal solution in a software system, and other studies show [125] that TD accumulation is not completely controllable and avoidable. Therefore, the classes and method proposed here help identifying and prioritizing ATD items that have more impact and therefore more interest to be paid. Such prioritization is important both for research and practice: for the former one, it points at classes of ATD and interest in need for further research and for tool support (for example, new metrics), which seems still quite scarce. As for practice, such prioritization would save the resources (often largely inferior than the cost of repaying all the TD present in the system) when deciding what to refactor. The impact analysis using the factors can be particularly useful as input in the prioritization mechanism in order for the managers to understand when an ATD item has to be refactored.

Predicting growth related to TD is the key prevention: The main prevention for contagious debt and in general for increasing interest, is to understand the growth of various factors: the growth of the system (in order to understand the growing interest on the principal), the testware, the complexity, the users and the NFRs connected to the injected ATD. This implication brings new knowledge in the field of software engineering, since it contrasts with previously derived theories on Technical Debt in which only maintainability is recognized as interest of an ATD item.

Technical Debt and Ecosystems: 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 (or *quarantines*) that would avoid the phenomenon, but also limiting the spread, once the contagious ATD item would be identified. On the other hand, ecosystems such as the Open Source community participating in company B highlights how such development collaboration would actual have a positive impact on avoiding costly TD accumulation thanks to the continuous external pressure and focus on internal qualities of the software.

Combination of linear accumulation with linear interest might lead to non-linear interest: According to another study [125], 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 [125], 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*, 8.5.2). The implications of blending the models in this paper with the ones in [125] 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.

8.7.2 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 by comparing data coming from multiple companies, multiple sessions and multiple cases across several 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 (eight) 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 8 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. In the case of Contagious Debt, we validated the by collecting 12 cases explicitly reflecting such phenomena.

8.7.3 Related work

The phenomenon of ATD has only recently received the attention of the research community [33][38]. 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. [35]) or code-smell [128]. 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. [124], 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 [129], and the assumption, in more than one work (e.g. [39], [124]) 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 [125], 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 [39]. 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. [36],[46],[126]. In [36] 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.

8.8 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 8 sites at 6 large international software companies, the most dangerous in terms of generated effort. We have provided a map of the effects and the related activities (which represent the interest) that can be connected with the ATD classes in the previously mentioned taxonomy. Such results would make aware the practitioners and would help researchers in future directions. The paper shows newly discovered patterns of socio-technical events such as *contagious debt* and *vicious circles,* which cause the continuous increment of interest to be paid over time. Such increment has been empirically studied in 12 concrete examples of ATD items: we provide an understanding of the factors that are involved in increment of the interest. These factors can be proactively monitored in order to strategically refactor the ATD before it becomes too late to be repaid. The results reported here contribute to the field of empirical software engineering by providing clear goals for which measures are needed in order to support the software development (according the goal-question-metrics approach), and by updating the theoretical framework for technical debt thanks to new empirical evidence, which show how the interest is composed by several factors and cannot be restricted to maintenance effort only. The results in this paper can be used in research and industry in order to create and improve practices dedicated to monitor, uncover and prioritize the dangerous ATD present in the system. This way we help practitioners in recognizing and proactively avoiding situations where refactorings are not completed or ATD items become "contagious", which would benefit the companies itself but would also avoid the pandemic spread of the ATD in entire ecosystems.

9 EVALUATION OF ARCHITECTURE TECHNICAL DEBT INFORMATION FOR BALANCING AMBIDEXTERITY

Architectural Technical Debt is a metaphor for representing sub-optimal architectural solutions that might cause an interest, in terms of effort or quality, to be paid by the organization in the long run. In this Chapter we evaluate if such metaphor is useful for communicating risks of suboptimal solutions between technical and non-technical stakeholders. It's fundamental to understand the information needs of the involved stakeholders in order to produce technical debt measurements that would allow proper communication and informed prioritization. We have investigated, through a combination of interviews, observations and a survey, what key information is needed by agile product owners and software architects in order to prioritize the refactoring of risky architectural technical debt items with respect to feature development. These results show how the information developed in the previous chapters is useful for the stakeholders and requires to employ part of the development in order to trade short-term responsiveness with the long-term one.

This chapter has been published as:

Martini A., Bosch J.: *"Towards Prioritizing Architecture Technical Debt: Information Needs of Architects and Product Owners"* Accepted for publication in proceeding of Euromicro SEAA 2015 [130]

9.1 INTRODUCTION

Large software industries strive to make their development processes fast and more responsive, minimizing the time between the identification of a customer need and the delivery of a solution. The trend in the last decade has been the employment of Agile Software Development (ASD) [6]. 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 [38], [31]. Tom et al. [33] have explored the TD metaphor and outlined a first framework in 2013, and a recent systematic mapping has highlighted the current gaps in research [31]. Part of the overall TD is to be related to architecture sub-optimal decisions, and it's regarded as Architecture Technical Debt (ADT) [38], [31]. ATD items are considered as violations in the code towards an intended architecture for supporting the business goals of the organization [31]. An example of ATD might be the presence of structural violations [124].

According to a recent study carried out by the same authors of this paper [125], the prioritization of ATD with respect to feature development is an important activity in order to balance the short-term value delivery and the long-term responsiveness of a company. Not prioritizing ATD might lead to software development crisis followed by big refactoring activities preventing the continuous delivery of features [125]. Although such prioritization seems a critical activity, the recent mapping study found a gap in the current scientific knowledge: "More industrial studies are needed to show how to prioritize a list of TD items to maximize the benefit of a software project and which factors should be considered during TD prioritization in the context of commercial software development." [31]. We therefore intend to fill such gap by exploring how the prioritization activity is performed and what are the information needs between the key actors in such activity: the software architects and the product owners (*POs*). We

especially investigated large Scandinavian companies developing embedded software and employing Agile software development. Our research questions are the following:

RQ1: What is the information needed by product owners and architects to prioritize ATD with respect to feature development?

RQ2: What are the differences between architects and product owners when prioritizing ATD with respect to features?

We have employed a multiple case study involving 6 cases in 4 large companies. We have interviewed four kinds of roles, software, system architects, product owners responsible for single products and for portfolios. In order to shed light on what is the important information needed to prioritize ATD, we have collected different kinds of data, by qualitatively observing the prioritization activity and by collecting quantitative data from the participants with a questionnaire.

9.2 Background and Conceptual Models

In order to prioritize ATD items with feature development, we need to understand how they are compared in practice, what aspects are used for feature prioritization and which aspects are influenced by the information about ATD.

9.2.1 Features vs ATD refactoring prioritization model

Figure 46 describes our conceptual model. Such conceptual model is based on previous data collected of the same authors of this paper at the same companies involved in the study ([44]).

When a decision needs to be taken between refactoring ATD and developing new features, the main actors interacting in the prioritization activities are Product Owners (POs in ASD) and software architects. The interaction between the two roles leads to a decision to use resources for refactoring or for feature development, which in turn leads to the outcome of the activity, a development plan. Depending on the size of the ATD item, the plan can involve sprint planning, including items in the backlog for the Agile teams, or it might be on a broader scope, for example the definition of a roadmap.

The prioritization activity is based on different aspects taken in consideration by the decision makers. For aspects we mean "[…] a property or attribute of a project and its requirements that can be used to prioritize requirements" [131].

The prioritization activity is based on different information: the one coming from the customer (not covered in this paper), and the information about the ATD effects (for example, the cost of refactoring, or else the principle of the ATD, and the impact of the ATD, or else its interest). POs and architects take in consideration the input information and evaluate it according to the prioritization aspects. When all the important aspects are evaluated and weighted, the actors take a decision.

Our aim in this paper is to explore what ATD information is needed during the prioritization activity in order, for POs and architects, to be able to take a decision (RQ1). We also investigate how the prioritization aspects and ATD effects are weighted in different ways by POs and architects (RQ2).

In the next sections we define the prioritization aspects used for the comparison of weights between architects and POs, and the available information on ATD effects that can be used during prioritization. As explained in the methodology section, these concepts were used during data collection.

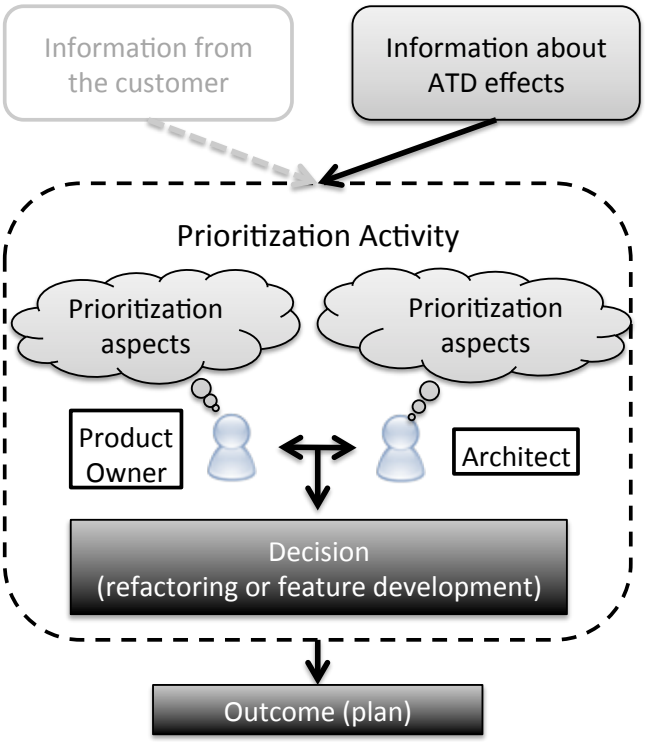


Figure 46. Conceptual model

9.2.2 *Prioritization aspects*

[131] is a literature survey about prioritization aspects currently considered important in literature and in industry. We have adapted such aspects to the ones used in the investigated companies, by aligning the definitions in [131], [132] with the ones given by the participants in the beginning of the data collection. This step is necessary, as also suggested in [131], "It is important that the stakeholders have the same interpretation of the aspects […]"

The aspects are:

- *Competitive advantage*: A superiority gained by an organization when it can provide the same value as its competitors but at a lower price, or can charge higher prices by providing greater value through differentiation.
- *Customer long-term satisfaction*: the ability of keeping the customer satisfied not only with short-term value but also with value that lasts. An alternative definition is "gaining the customer trust".
- *Lead time*: the ability to deliver the same amount of features in a shorter time.
- *Maintenance Cost*: the development effort spent to maintain a product.
- *Attractiveness for the market* (also *market attractiveness* in the following): the ability of delivering features that make the product attractive for new customers.
- *Penalty*: the penalty that is introduced if a requirement is not fulfilled (or a feature is not delivered).
- *Risk*: "Every project carries some amount of risk. In project management, risk management is used to cope with both internal (technical and market risks) and external risks (e.g. regulations, suppliers)" [131].
- *Specific customer value*: the ability of delivering features that have value for a specific customer rather than for all the customers.

- *Volatility*: rate of change over a given period. Volatility varies, for example: the market changes, business requirements change, legislative changes occur, users change, or requirements become more clear during the software life cycle [131].

9.2.3 Architecture Technical Debt effects

The same authors of this paper have recently investigated several cases of ATD, and found that there are effects of some ATD items for which the interest to be paid by the company was particularly high [42]. Such paper reports a taxonomy of this effects, that we have used in this investigation as the possible information about the ATD effects that can be used during the prioritization by product owners and architects (Figure 46). Since [42] is accepted for publication but is not accessible yet, we report the taxonomy here.

- *"Double" effort*: it's an effect generated by code duplication. The word "double" is only indicative of the extra effort that could be mapped double maintenance of the duplicated code. We cleared this term with the respondents, who interpreted the value as possibly being between the 25 and 100% of extra effort.
- *Big deliveries*: the large amount of dependencies among the components in one of the sub-systems cause, each time a new release involved a small change, the test of the whole sub-system instead. 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).
- *Contagious ATD*: it's the main problematic effect related to the large amount of dependencies. 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.
- *Developers idling*: the time spent by the developers in understanding parts of the system that are not familiar with and by understanding which pattern to use in similar situations.
- *Many code changes*: when the presence of ATD causes the adaptation of the existing code, which needs to be continuously changed.
- *Number/complexity of test cases*: when ATD causes the existence of more test cases or test cases that are more complex (and therefore more difficult to be managed).
- *Probable hidden TD*: see "contagious ATD" for the explanation.
- *Quality issues*: when an ATD is likely to create 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)
- *Wrong estimation of effort*: the unawareness of the hidden ATD and its ripple effects, 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.

9.3 RESEARCH DESIGN

9.3.1 Case selection

We have employed an embedded multiple-case study [63], 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 6. We selected, following a literal replication approach [62], 4 companies: A, B (3 sub-cases), C and D, large organizations developing software product lines, having adopted ASD and had extensive in-house embedded software development.

Company A is a manufacturer of recording devices. The company employed SCRUM, has hardware-oriented projects and use extensively Open Source Software. *Company B* is a manufacturer of telecommunication system product lines. They have long experience with SCRUM-based cross-functional teams. We involved three different departments within company B (B1, B2, B3). *Company C* employed SCRUM to develop a product line of devices for the control of urban infrastructure. *Company D* is involved in the automotive industry. The development in the studied department is mostly in-house, recently moved to SCRUM.

9.3.2 Data collection

We have collected data using two complementary methods: we ran a questionnaire and we complemented it with qualitative interviews with the same participants. This way, we have collected both qualitative and quantitative data, while we have complemented group thinking with individual responses. This strategy was used in order to achieve *method triangulation*, as suggested in [63].

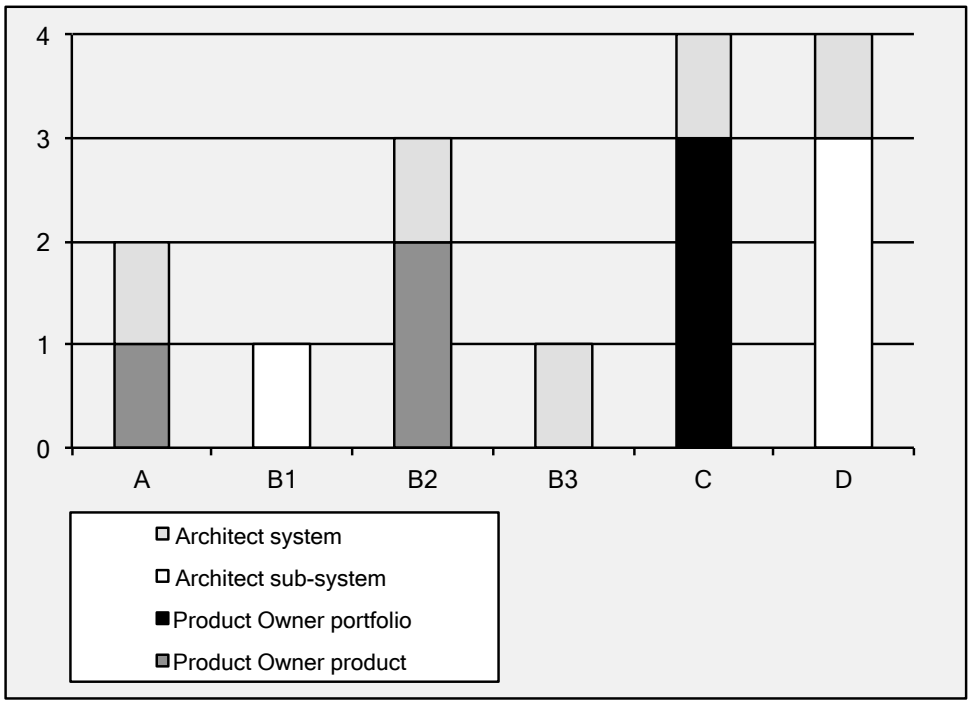


Figure 47. Distribution of roles and companies

The roles of the participants are outlined in Figure 47. In total, we had three portfolio managers, three product owners, five architects responsible for a system level and four architects responsible for sub-systems. This combination assured that all the roles were covered.

The workshop consisted of the following phases:

1 – Preparatory workshop for prioritization aspects: before the investigation, we spent 30 minutes in aligning the prioritization aspects used during the following activities among the researchers and the participants from the companies. All the participants were present in the preparatory workshop.

2 – Questionnaire on the prioritization aspects: we asked the practitioners to rate the prioritization aspects. The main question was:

Which Prioritization Aspects do you use for prioritization?

We then asked the practitioners to rate each aspect described in the background section (*Competitive advantage*, *Lead time*, etc.) according to three parameters:

- *Importance*: how important is this aspect in the prioritization activity. We asked the participants to rate the aspect using the following Likert scale: 1 – Low, 2 – Medium/Low, 3 – Medium/High and 4 – High.
- *Frequency*: how often this aspect is used. We asked the participants to rate the aspect using the following Likert scale: 1 – Seldom, 2 – Sometimes, 3 – Frequently and 4 – Always.
- *Granularity*: how precise are the values used during prioritization. 1 – Coarse, 2 – Medium/Coarse, 3 – Medium/Fine and 4 – Fine. The *Fine* granularity was mapped to precise measures, while the *Coarse* one was mapped to expert judgment not based on measurable data.

This way we covered different dimensions of the same aspects. In the results, we will report both the single value and a unique value for the average between importance and frequency for each aspect: such index shows the overall contribution of such aspect during the prioritization. The granularity shows how precise data is used in the prioritization.

3 – Focus groups on the prioritization cases: in this phase we split the participants in two focus groups, where we analyzed two concrete cases. The cases were presented by company A and company C and were considered difficult to be prioritized when deciding between refactoring and feature development. In each session, lasted one hour, a case was reported by a participant from the related company and one researcher was present to guide the relevant prioritization aspects to be covered. The other participants were split in two groups, and the researchers made sure that there was an even number of participants for each company and for each role in each group. Each group then discussed the case with respect to the prioritization aspects explained previously.

4 – Plenary summary session and ATD effects analysis: after the parallel focus groups, the participants were gathered together. Both the groups summarized their results to the other group. Then, the researchers introduced the concepts of ATD, the ATD effects and aligned such concepts across the group. Then, the ATD effects were mapped to the analysis done on the cases. The participants discussed if the information about the ATD effects would be useful for the prioritization activity in the analyzed cases and in other examples.

5 – Questionnaire on information needs about ATD: after the plenary session, we asked the participants to answer the following questions for each of the ATD effects mentioned in the background section (*double effort*, *contagious ATD*, etc.).

- *Would you use this attribute in the prioritization?* – we asked the participants to answer one of the following Likert-scale options: 1 – Very Unlikely, 2 – Unlikely, 3 – Likely, 4 – Very Likely.

- *Which granularity (precision of the value) would you need?* – we asked the participants to answer one of the following Likert-scale options: 1 – Coarse, 2 – Medium/Coarse, 3 – Medium/Fine and 4 – Fine.
- *How useful would be the ATD information during prioritization?* – Possible answers were: None, Low, Medium/Low Medium/High, High.
- *How many resources would you allocate to obtain the information on the ATD?* – None, Low, Medium/Low Medium/High, High. We also asked the participants to specify a percentage related to the development effort. We agreed with the participants, before answering the question, that the scale could be mapped in the following way: 0% – None, 10% – low, 20% – Medium/Low, 30% – Medium/High, 40% – High.

We also asked the participants to map the effects to the prioritization aspects: this way, we could understand which aspect the information from the ATD effects would be useful for. More specifically, we ask the respondents to map, for each prioritization aspect, the three most relevant ATD effects that would give more information for evaluating such aspect.

9.3.3 Data Analysis

Quantitative analysis – we have analyzed the answers from the questionnaire in a quantitative fashion, i.e. by interpreting the numbers obtained from the answers. Given the small sample, we could not use any statistical method. However, several numerical comparisons could be performed, which helps understanding the role-specific perception of the prioritization aspects and the information needed from the ATD analysis.

We have done three kinds of quantitative analysis, and the results will be presented following the same structure:

- Overall results – we have considered all the answers together and we have used such results to draw conclusions common to all the roles and companies.
- Role-specific results – we have divided the results as *product owner* view and *architect* view. Dividing the answers also by level of abstraction (system/software architects and product/portfolio owners) would have reduced the amount of data for each role too much.
- Comparison – we have compared all the results coming from the different roles in order to understand if there were differences and about what.

When interpreting the answers the Likert scales, we have weighted the difference of each option with a unitary value. For example, for the answers on the importance of each prioritization aspect we have weighted them as “Low” = 1, “Medium/Low” = 2, “Medium/High” = 3 and “High” = 4. This was done in order to have similar intervals to be compared. To the questions that were non-answered we assigned a value equal to 0, as we specified several time with the respondents.

Qualitative analysis – We have qualitatively analyzed the discussions in order to complement the questionnaire data and to understand which factors would influence the discussion between the different roles. We have analyzed the case discussion in order to understand which information was more requested by the different roles and if there were conflicts.

9.4 RESULTS AND ANALYSIS

We will present our results by showing the questionnaire data for each question in the three different perspectives mentioned in the previous section: *overall*, *role-specific*

and *comparison*. In some cases, when possible, the findings will be condensed in a unique table. We have colored the cells in order to make easier for the reader to compare the values in the cells. We will offer, together with the results, the researchers' interpretation of the data based also on the prioritization cases discussed during the workshop.

9.4.1 Prioritization Aspects

Table 25 .Rank of Prioritization Aspects

Aspect\Rank	Imp.	Freq.	Avg.	Gran.
Competitive advantage	3.4	2.7	3.05	1.9
Specific customer value	2.9	2.3	2.6	2.4
Market attractiveness	2.5	2.3	2.4	1.8
Lead time	2.4	2.3	2.35	1.7
Maintenance Cost	1.8	2	1.9	1.4
Customer long-term sat.	1.8	1.8	1.8	1
Risk	2	1.5	1.75	1.2
Penalty	1.9	1.3	1.6	2
Volatility	0.2	0.3	0.25	0.3

The overall rank given by the respondents about the prioritization aspects is shown in Table 25. We show the rank of the importance, frequency, and their average (and the aspects are sorted by using this attribute). We also show, in the last column, the granularity currently used with each aspect.

Table 26 shows the ranked averages (calculated as in the *Avg.* column in the previous table) for the architects, the POs and the difference in their ranking. The table is sorted using the absolute value of the difference (not shown). The sign of the *Diff.* value shows, if < 0, that the POs have ranked this aspects more important that the architects, while if the *Diff.* value is > 0 the opposite statement is true.

From the analysis of these tables it's quite clear that the POs use, when prioritizing, the specific customer values and the attractiveness for the market of the products more than the architects. Also *lead time* and *competitive advantage* show the same differences, even though with less emphasis.

Table 26. Comparison of ranks between POs and architects

Aspect\Rank	Arch.	POs	Diff.
Market attractiveness	2.05	3.1	-1.05
Specific customer value	2.15	3.1	-0.95
Lead time	2.05	2.7	-0.65
Competitive advantage	2.85	3.5	-0.65
Customer long-term sat.	1.85	1.5	0.35
Penalty	1.35	1.7	-0.35
Maintenance Cost	2.05	1.8	0.25
Risk	1.7	1.5	0.2
Volatility	0.1	0.3	-0.2

9.4.2 *ATD effects usefulness in prioritization*

Table 27 shows the overall rank of the usefulness of the ATD effects for prioritization, the architects' and POs' rank and their difference. The table is sorted by the absolute value of the difference (not shown).

Table 27. Ranks and comparison of ATD effects

Effect\Rank	Over.	Arch.	POs	Diff.
Big deliveries	2.3	2	2.8	-0.8
Many code changes	2.5	2.8	2	0.8
"Double" the effort	2.8	2.7	3.2	-0.5
Number/compl. of tests	2.5	2.7	2.2	0.5
Quality issues (bugs)	3	3.2	2.8	0.4
Wrong effort estimation	2.5	2.3	2.6	-0.3
Contagious ATD	2.8	2.8	3	-0.2
Probable hidden ATD	2	2	1.8	0.2
Developers idling	1.9	1.8	1.8	0

The table shows that the three top important ATD effects to be known for prioritization are *Quality issues*, *"Double" the effort* and *Contagious debt*. The less important ones seem to be *Probable hidden ATD* and *Developers idling*.

From the *Diff* column we can see how the POs are more concerned with big deliveries than architects, while the architects are more concerned with code changes. POs and architects agree on considering less important *Probable hidden ATD* and *Developers idling*, while they agree on considering the information about *Contagious ATD* as quite important for prioritization purposes. It's interesting to see how *Quality issues* (bug fixing), is considered more important by the architects than by the POs.

Table 28. Mapping of ATD effects to prioritization aspects

Effects\ aspects	P	M	LT	R	V	C	S	A	LS	Tot
"Double" effort	4	8	26	2	0	0	0	0	0	40
Big deliveries	0	3	14	10	0	0	0	0	0	27
Code changes	1	7	9	21	0	0	0	0	0	38
Num/compl. tests	0	11	13	4	6	0	0	0	0	34
Quality issues	0	15	10	10	0	4	2	1	2	44
Hidden TD	3	8	9	11	0	0	0	0	0	31
Wrong estim.	12	1	20	0	0	1	2	0	0	36
Contagious ATD	5	9	10	13	3	0	0	0	0	40
Developers idling	2	9	14	3	0	3	0	0	0	31
Total	27	71	125	74	9	8	4	1	2	

Table 28 shows the connection between the ATD effects and the prioritization aspects (identified by their initial letters: *P = Penalty*, *M = Maintenance*, etc.). The numbers shown in the table are the sum, for all the participants, of the scores associated with the effect when it was ranked as usefulness information for a certain prioritization aspect. For example, *Double effort* has been ranked by 8 respondents as first most important information (weighed 3) influencing lead time (*LT*), second most (weighted 2) important for 1 respondent, and no respondent has used it as third most important

(weighted 1). Applying the weights, we have 8[respondents] * 3[weight] + 1 [respondents] * 2 [weight] + 0 [respondents] * 1 [weight] = 26.

The total in the bottom is the sum of the column values: such row shows how much information, according to the participants, an ATD effect gives when prioritizing using a given aspect. For example, the information about the ATD effects can be used quite a lot when considering the lead time aspect (*LT,* 125), while the ATD effects don't seem to be useful for prioritizing based on attractiveness of the product for the market (*A,* 1). The *Total* on the right shows how much each ATD effect contributes to the overall prioritization activity.

From this table, we can see how the studied effects can be especially useful when prioritizing based on lead time (*LT*). They are also quite useful when the maintenance costs (*M*) is taken in consideration during prioritization and when risk is considered an important aspect (*R*).

As for the comparison between the roles, we report the most relevant results without reporting the tables, for space reasons. We have selected the responses that show conflicts between the architects' and the POs' views. The architects don't consider *big deliveries* as impacting lead time, whereas POs consider it quite an influencing effect. Architects consider *contagious debt* as more useful when prioritizing considering the *lead time* and *penalty* aspects, while POs consider it more useful when considering the *risk* aspect. In general, architects consider the ATD effects more useful for prioritizing based on the *penalty* and *maintenance cost* aspects than POs.

Table 29 Usefulness of ATD information

Answer\Role	Arch.	POs	Total	%	Diff.
Low	0	0	0	0.00%	0
Medium Low	0	1	1	12.50%	1
Medium High	1	2	3	37.50%	1
High	3	1	4	50.00%	2

Table 29 shows the answers to the question "How useful would be the ATD information during prioritization?". We show the partial answers divided by the roles and the total answers, also presenting the percentage of answers and the difference between the roles. The table confirms that almost all the respondents considered ATD information quite useful for prioritization; however, architects seem more convinced of its usefulness than the POs.

shows the answers to the question "How many resources would you allocate to obtain the information on the ATD?". As explained in the methodology section, *Low* corresponds to 10%, while *High* corresponds to 40% of the development time.

Table 30. Recommended resources for ATD information

Answer\Role	Arch.	POs	Total	Percentage
None	0	0	0	0.00%
Low	1	1	2	25.00%
Medium Low	3	3	6	75.00%
Medium High	0	0	0	0.00%
High	0	0	0	0.00%

We can see how POs and Architects agree on the amount of resources needed for providing information about ATD effects. 75% of them would allocate 20% of the development time, whereas the remaining 25% would allocate 10%.

9.5 DISCUSSION

In this section we will discuss our contributions with respect to research and practice. Then we will discuss limitations and threats to validity and we will compare our results with the related work on this subject.

9.5.1 Implications for research

Most of the existing literature concerning ATD is related to the identification of items. As pointed out in a recent systematic mapping [31], there has been little work on the prioritization of ATD and most of it is on a theoretical level [36], [40] and "More industrial studies are needed to show how to prioritize a list of TD items to maximize the benefit of a software project and which factors should be considered during TD prioritization in the context of commercial software development". We have started to fill this gap with this exploratory study, especially for what concerns prioritizing ATD in large companies developing embedded systems and with an Agile setting in place.

The exploratory findings show that the information on ATD is regarded as quite important both from architects and POs for practical prioritization of refactorings with respect to feature development, especially when taking in consideration lead time. We provide an important and missing piece of the puzzle in current literature, since no studies investigate how applicable would be in practice the information about various ATD items, especially with respect to a key stakeholder such as the business side of the organization: the POs.

Furthermore, our results show a prioritization of the key ATD effects to be further investigated in research because considered important for prioritization. Especially important seems to be the understanding of the *contagious debt* phenomenon [42], the ATD items related to *"double" effort*, such as duplication and reuse, and to *quality issues* such as a high number of defects to be fixed and the time spent by the developers on finding the source of the issues.

The results highlight how the *risk* aspect is quite difficult to be considered during the prioritization activity, because of the expensiveness to produce evidences and the lack of certainty about risk, and because of the difficulty for the stakeholders to compare the values with the other prioritization aspects (as shown in one of the cases analyzed). Further research is therefore necessary in this area in order to understand how practical *risk* measures can be retrieved and automated from existing artifacts and how such data should be interpreted.

We recognize a gap in the existing ATD effects: almost none of them are useful to prioritize when considering competitive advantage and attractiveness of the product on the market. However, such connection should be more investigated in order to understand if ATD is not related to such aspects or if there has not been enough research on such connection.

With this study we have provided an exploratory investigation of the research problem: the design can be reused in research in order to gather further evidence coming from larger samples and involving different kinds of companies. The same authors of this paper are in the process of designing a larger study involving many more participants, both POs and architects, from several companies in order to strengthen the current results with a quantitative investigation.

9.5.2 *Implications for practice*

The paper shows how the practitioners recognized the importance of the information about ATD effects. Practitioners from other similar contexts can use the results currently available in literature for better prioritizing between refactorings and feature development. We offer a taxonomy of the ATD effects that need to be retrieved and a map (Table 28) of such effects to the aspects that they can be used for prioritization.

The results in Table 30 can be useful for allocating resources in order to collect company-specific data on current ATD. The current (and quite consistent) recommendation, from the practitioner is to allocate between 10 and 20% of the resources in order to take care of technical debt.

The information needs recognized in this paper can be used in order to collect company-specific measures on ATD. The knowledge about the information needs of the main stakeholders (architects and POs) is regarded as a prerequisite for the development of reliable and useful metrics, as defined in the ISO standard for the development of quality models [41].

9.5.3 *Limitation and threats to validity*

This study has the following limitations. The most important limitation is the limited sample size of the respondents. However, we aimed at reporting the understanding from a case study, which included both qualitative and quantitative data, rather than surveying a large number of practitioners. Furthermore, as previously explained, we offer a novel understanding of an important and overlooked research subject. The authors aim at collecting such evidence, using an investigation tool improved with the help of the exploratory results obtained from the study presented here.

As for the threats to validity in case studies, we refer to the ones mentioned in [63]: construct, internal, external validity and reliability. We mitigated the threat to *construct validity* (concerning the validity of the investigation device) by aligning the concepts used during interviews and questionnaire with a preliminary workshop in which all the participants were present. The researchers were also present during the participants' activity of answering the questionnaire, which helped clarifying possible misinterpreted issues during data collection. In this study we did not try to address *internal validity* threats, since we did not investigate causal relationships, and therefore we did not need to investigate the existence of possible factors influencing such relationships. However, we have investigated the ATD effects and the prioritization aspects that other studies have identified as the currently most relevant: other, less recognized aspects and effects could be found to be as relevant as the ones investigated in this study. In order to mitigate possible threats of *external validity*, we have investigated several roles from several companies. We restrict our claims to the investigated context, large companies developing embedded software, employing ASD and developing a line of products. Also, our companies were located in the Scandinavian area, which might influence the results with the presence of a cultural background that might not be present in other contexts and other countries. Finally, we supported the *reliability* of our results by employing three kinds of triangulation: *observer triangulation* (both the researchers were present in the data collection and analysis), *method triangulation* (we collected quantitative and qualitative data) and *source triangulation* (we collected data from four different companies and 4 different roles). We have also reported quite extensively here the questionnaire design and results, so that other researchers can replicate this study.

9.5.4 *Related work*

As mentioned by a recent systematic mapping [31], there is a need for more industrial studies on how prioritization of ATD is carried out. The paper [36] is focused on the theoretical prioritization of generic TD items among themselves, as well as

[133], which is focused on design debt. Such results needs a next step in which the theoretical model is put in practice with real and context-specific items. Also, such items need to be put in contrast with the features prioritization, which is the main cause for down-prioritizing ATD (the objective of our study). In [40] the authors propose first a formal approach for defining the TD as *evolution steps* and *technical debt items*. Although the approach gives a solid theoretical foundation to the decision making about TD, such approach still suffers from several shortcoming (recognized by the authors themselves) when used in practice: this approach is only based on the cost, and it does not take in consideration other prioritization aspects (that we have investigated here), proper of the prioritization of TD when compared to features. The authors recognize the need of further research on this path "[…it would be interesting] if we extend the metrics beyond development cost to customer benefit, training costs, time to market and similar criteria". We have done a first step to map the ATD effects, for which metrics are being developed ([42]) to some of such aspects. In [134], the authors survey software refactoring: while they take in consideration the prioritization of refactorings among themselves, the section *Identifying where to Apply which Refactorings* does not take in consideration the perspective of POs and feature development. Our results are in line with the findings, in [26], that recognize how (generic) TD is used as a prioritization aspect in Agile. However, such study does not investigate in depth how information on ATD effects is used in the prioritization activity. Our findings also confirms that risk is quite a difficult aspect to be taken in consideration for prioritization and that further research is needed on the subject [26]. Also, our mapping shows how information about ATD effects would be useful for using the *risk* prioritization aspect.

9.6 CONCLUSIONS

The prioritization of ATD with respect to feature development, a critical issue for assuring constant value delivery, is currently overlooked in research and represents a struggle for software companies. We have done a first step towards understanding the information needs of the main actors involved in the prioritization of ATD refactorings and feature development. We have investigated which prioritization aspects are most relevant in the prioritization activity: the results show that some important aspects, such as *lead time, maintenance costs* and *risk*, would largely benefit from the information related to the ATD effects, for which we present a mapping tool 0We have also highlighted how measures of ATD effects, especially *contagious debt*, *quality issues* and *"double" effort* would be strongly appreciated by architects and POs developing software for 4 different large companies. The respondents would dedicate between 10 and 20% of the resources in order to manage ATD. Finally, we highlight the similarities and differences in evaluating different ATD effects and different prioritization aspects by POs and architects, for example their different concerns about *big deliveries* and *maintenance costs*. The next step, already in the authors' plan, consists of conducting a version of this investigation involving a large sample of respondents from a large set of companies, in order to further strengthen the current case study-specific findings with a broad, quantitative investigation. Another important next step is to analyze the customer-related information used during the prioritization activity in order to understand how they are compared to the ATD effects in order to prioritize ATD refactoring against feature development.

10 THE CAFFEA FRAMEWORK AND THE ORGANIZATIONAL SOLUTION FOR AMBIDEXTERITY MANAGEMENT

In this Chapter we study a solution in order to achieve responsiveness both in short and long term. A major concern is how to continuously develop and manage the reference architecture shared by the Agile teams. Through empirical investigation of 3 sites from 2 large product line companies developing embedded software and the conduction of analysis based on the Grounded Theory approach we have developed an organizational framework, CAFFEA, for continuous architecting. The framework has been statically validated through a cross-company workshop including participants from 7 sites from 5 large software companies, discussion groups and a final survey.

The organizational framework includes (virtual) teams that need to be put in place to take care of architecture activities overlooked in the Agile organizational shift. The responsibilities for the activities have been mapped to key architect roles needed to support the Agile teams. We have validated the results by a cross- case workshop combined with a survey. The organizational framework has been recognized as sound to be implemented.

A short version of this chapter has been published as:

Martini A., Pareto L., and Bosch J., *"Towards Introducing Agile Architecting in Large Companies: The CAFFEA Framework,"* in Agile Processes, in Software Engineering, and Extreme Programming, 2015. [135]

10.1 INTRODUCTION

Large software industries strive to make their development processes fast and more responsive, minimizing the time between the identification of a customer need and the delivery of value. Short term responsiveness is given by Agile Software Development (ASD) [6]. Long-term goals and customization of the products for different customers need to take advantage of software product lines, which relies on a platform capable to support quick value delivery of new features or products, built on shared asset. Few and mostly theoretical exploratory studies have been carried out to find out if ASD can be applied to large software product lines [71], [72], [136], but definitive conclusions with practical implications are still difficult to be drawn.

Recent research shows how the accumulation of architectural technical debt [137], [33], (the presence of underdeveloped or quickly eroded architecture) might lead, in the companies, to development crises [125], blocking long-term value delivery. A gap in the current Agile frameworks is the lack of activities to enhance agility in the task of developing and maintaining software architecture (*Agile architecting*), necessary for long-term responsiveness [6][26]. The role of architects becomes crucial, but there is a lack of knowledge, in literature, on how such roles are implemented in ASD. Therefore, the research questions that we want to inform are:

RQ1 What are the challenges in conducting architecture practices in Agile software development employed in large software product line organizations?

RQ2 Which roles and teams are needed in order to mitigate the challenges in conducting architecture practices in large product line organizations employing Agile?

We have combined literature review, interviews involving several roles in large product line companies employing Agile Software Development and a combination of structured inductive and deductive analysis in order to find the gaps in the architect roles and their activities. We have developed an organizational framework, *CAFFEA*

(Continuous Architecting Framework For Embedded software and Agile), comprehending roles and teams to address the challenges related to the architecture practices. The framework has been preliminary validated, both statically by 40 participants from 7 companies and dynamically after its introduction in 4 companies.

The model can be used as guidance for defining roles and teams in a large product line organizations employing ASD, with the aim of supporting Agile architecting. The contribution of the study is threefold:

- We show the current gaps for managing software architecting in ASD employed in large product line companies.
- We present an organizational framework (CAFFEA), composed by roles, teams and practices aimed at addressing the gaps found.
- We report on the introduction of such framework in 4 companies, and on the preliminary evaluation of its application.

In the next sections we show our research design, in section 3 we present the results, in section 4 we discuss validity of the study, and finally in section 5 we discuss our main contributions to research and practice together with limitations and related work.

10.2 RESEARCH DESIGN

Our research design is visible in Figure 48.

10.2.1 Case Selection

We have employed an embedded multiple-case study [63], 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 [62], 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 [62] (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.

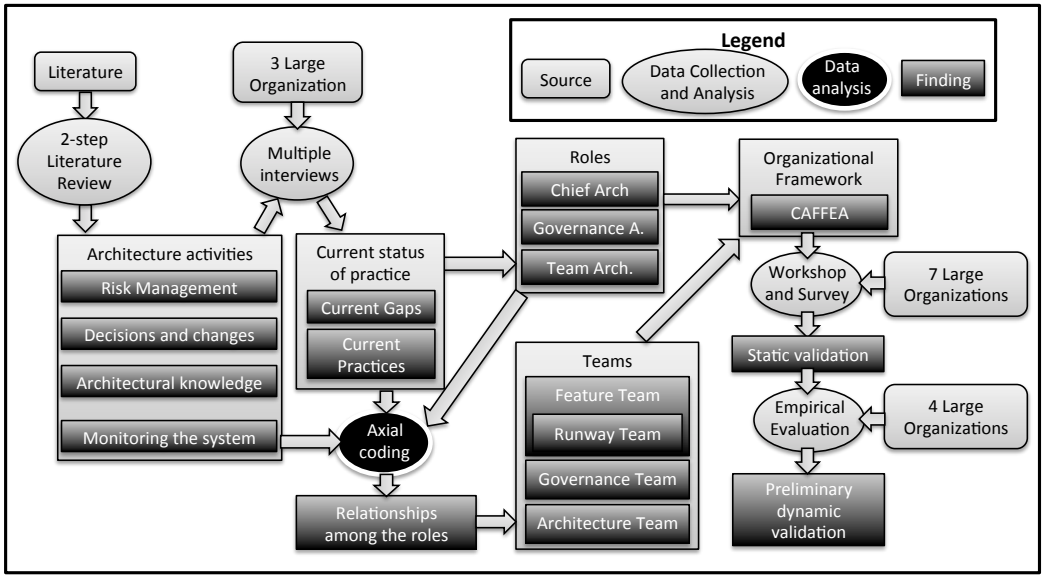


Figure 48. Research Design.

10.2.2 Data Collection

Step 1 – Literature review – We surveyed the literature in two steps, to collect the different activities carried out by the architects in practice. We selected [43] as an up-to-date (2008) and comprehensive categorization of "*what do software architects really do*". From such classification, we conducted a literature review for each class of practices, selecting the articles containing condensed knowledge [10]–[19].

Step 2 – Empirical mapping – We conducted 3 in-depth sets of interviews involving 3 of the cases, in particular A, C_1, and C_2. The interviews lasted 4 hours and involved developers, testers and architects responsible for different levels of architecture (from low level patterns to high level components). In total we collected 12 hours of interactive workshop discussions involving 25 employees.

During the interviews we assessed if the architecture practices found in step 1 were carried out, who was responsible, and what challenges they were facing. With this step we identified the current gaps in ASD with respect to architecture management.

Step 3 – CAFFEA Static Validation – After the development of CAFFEA using the previous data collection, we statically validated CAFFEA with a one-day workshop including 40 employees from 7 organizations. The validation workshop included a plenary session in which the researchers described CAFFEA in details (3 hours), a survey including a selection of 16 participants and a follow-up group discussion. For each question in the survey we used a Likert scale, ranging from 1 to 6. We have calculated median for the central tendency of the answers (in order to down-weighting outliers) and standard deviation to assess the polarity in the answers.

Step 4 – CAFFEA Preliminary Dynamic Validation – after the presentation of the previous results, some of the companies decided to implement CAFFEA. After 6 months, we set up a workshop with 4 of the companies to investigate which practices and roles were put in place and which benefits and challenges were found so far.

10.2.3 Data Analysis

The interviews and workshops were recorded and transcribed. The analysis was done following an approach based on Grounded Theory [55], alternating structured inductive and deductive techniques (described below) and using a tool for qualitative analysis, to the trace the code to the quotations.

Open Coding (inductive) **–** We analyzed the data in search for emergent concepts following open coding, which would bring novel insights, such as the identification of architect roles present in the organizations (after step 2).

Axial Coding (inductive) **–** The activities and roles were analyzed through axial coding in order to highlight *relationships among the roles* (Figure 48), which mapped the roles into the teams (right parts of Figure 48, after step 2).

Deductive Analysis **–** During step 1, we used the taxonomy of the activities in Kruchten [43] to analyze the papers from the literature review. For step 2 we used such approach when mapping practices and gaps to architect roles. As for step 3 and 4 (validation), we used practices, roles and teams both in the survey and questions.

10.3 RESULTS

First we show the identified architect roles in the companies, highlighting the challenges connected to such roles. We have divided the challenges in 4 groups: *risk management*, *architectural decisions and changes*, *providing architectural knowledge* and *monitor the current status of the system*. Then we present the teams, the organizational mechanism to address the challenges involving more than one role. The overall components and framework CAFFEA is visible in Figure 49.

10.3.1 Architect Roles

For each role, we first give a description of the role and we map it to the typical employee in the studied organizations, highlighting context differences when present. Then, we list the challenges related to such role.

10.3.1.1 Chief Architect (CA)

The main role of the CA is to take high-level decisions and to drive the rest of the architects and the Agile teams in order to build an architecture able to support strategic business goals. In all the organizations that we have studied, the role of CA is present and well recognized, and there are few challenges related to ASD.

Risk management

The CA is usually not directly involved in the detailed development: however, in order to take decisions on feasibility and to assist the sales unit with technical expertize, the CA needs to elicit the information about the current status of the system. The current challenge is the lack of such reliable information and therefore the risk of taking business decisions based on wrong assumptions made on the system.

Monitoring the current status of the system (communication input)

As mentioned before, the current communication practices lack good mechanisms for providing input to the CAs to take informed decision and to address past erroneous decisions (e.g. tool chains not working as expected).

10.3.1.2 Governance Architect (GA)

We found that the key for the scalability of Agile architecting in a large setting is an intermediate role between the CA and the teams. Such role, (we called it Governance Architect, GA) functions as a coordinator and support, giving strategic directions for a group of Agile teams developing features within the same (sub-) system. Many architecture practices were mapped by the informants to this role as the main responsible, and we found many challenges in the current organizations. Such role is not always formally recognized: this causes lack of coordination among isolated teams, which favors the accumulation of architectural debt. Also, the non-recognition of this

role leads to the lack of resources allocated for carrying out the needed architecture practices.

Risk management

The prioritization of short-term and long-term goals in the team is done by Product Owners through the backlog of the teams. However, such risk management activity usually leads to the down-prioritization of refactoring and architecture improvements, especially the long-term ones. A GA is needed to participate in prioritization to balance the focus between feature development and the long-term goals.

Managing decisions and changes

The architecture needs to support several features and the safe cooperation of the Agile teams. The investigation highlighted either the lack of such responsible for inter-feature architecting or the lack of communication and cooperation between the GA and the Agile teams.

Providing Architecture Knowledge (communication output)

With the shift to ASD, in some of the organizations (C_1 and C_2) the teams have changed from "component teams" to "generalized teams", free to change any part of the code given a feature to be implemented. However, such approach caused, in the teams, a lack of deep expertize about the components. The role of GA becomes therefore critical for assisting the teams and maintaining the architecture, both with face-to-face communication but also supported by documentation when the architecture knowledge is complex and extensive. The purpose of documentation is not to specify in detail the whole (sub-)system, but especially the critical points in which features and teams might hinder each other during the implementation (see "Managing decisions and changes"). An example mentioned by the respondents is describing mechanisms to access data resources, shared by different features. Such practices are quite time consuming and communication-intensive, and challenges have been reported in allocating resources.

Monitoring the current status of the system (communication input)

One of the most emphasized challenges during data collection was the accumulation of architectural debt [125]: the implementation in the code quickly drifted away from the architecture defined and used for strategic decisions and risk management by the CA and other management activities. GAs drive the monitoring and reacting to architecture erosion and need for evolution, together with the support of TAs (see "Team Architect") in the Agile teams. However, we found several gaps in all the organizations, especially the lack of iterative architecture consistency between implementation and the design/architecture, the non-feasibility of costly code review (growing code size with respect to the number of GAs), and the lack of resources employed in automated tools to help monitoring the erosion.

An important requirement for the GA, as pointed out by the respondents, is the combination of high technical skills, domain expertize and a communicative and charismatic personality able to lead and coach the teams.

10.3.1.3 Team Architect

The TA, the responsible for the architecture in the FT, is often present in the current organizations in the form of a technical leader or experienced developer. Such role is however not formally recognized, which bring the lack of responsibilities for the architecture practices in the teams.

Risk management

A challenge was the lack of participation of the team in risk management activities, such as tracking and reporting risky technical debt accumulated during the iterations (activity led by the TA) or to represent the interest of the teams in feasibility discussions with CA, GA and Product Owners (participation of TA).

Providing Architecture knowledge (communication output)

As mentioned for the CA and GA, the lack of capillary spread of architecture knowledge need to be mitigated by a peer in the team, which has been identified with the presence of TA, who would transfer the architectural knowledge from GAs.

Monitoring the current status of the system (communication input)

We found a lack of responsibilities, in the team, about tracking and reporting the status of technical debt that might affect other FTs. The TA would cover such responsibility, as well as lifting proposals for architecture evolution.

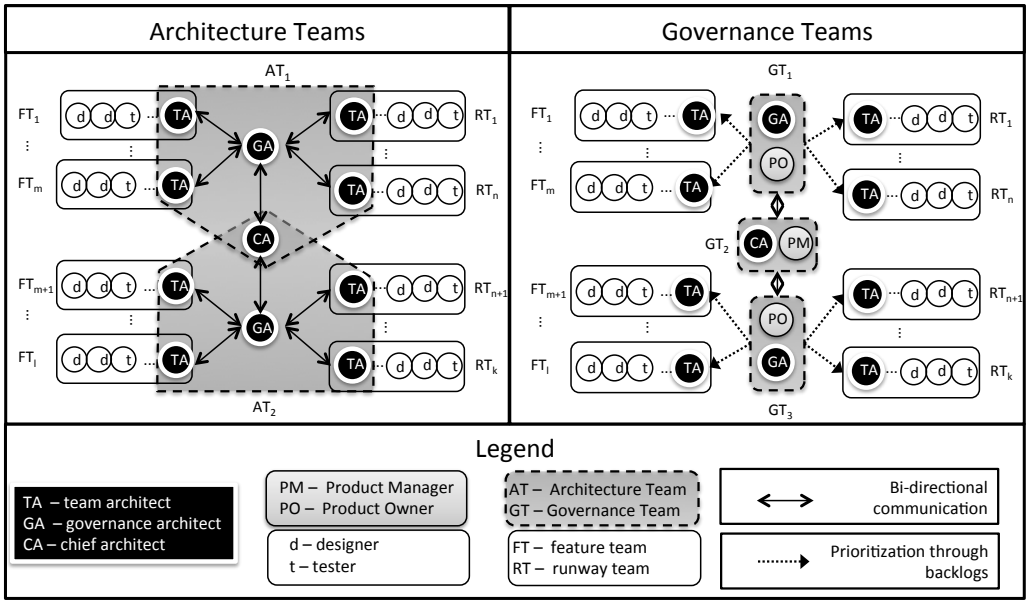


Figure 49. The components of CAFFEA: teams, roles and their relationships.

10.3.2 Teams

Analyzing the current gaps and the relationships among the architect roles previously mentioned, we found that most of the practices need the roles to coordinate and cooperate in order to mitigate the challenges. To achieve such coordination, suitable organizational mechanisms are non-permanent teams responsible for such practices visible in Figure 49. A special case is the Runway Team (RT), which involves a whole Agile team (see next section).

10.3.2.1 Runway Team

As mentioned about the GA and also confirmed by [125], a challenge in the studied companies is the down-prioritization of long-term refactorings or architecture improvements, causing the constant accumulation of architectural debt leading to responsiveness crisis. Such refactorings cannot be prioritized as stories in the backlog of the Agile teams, and therefore remains excluded from the development. In order to conduct such refactorings, a whole Agile team needs to be dedicated for one or more sprints to focus on the "architecture feature" rather than on customer-related features. We called such team "Runway Team" in order to recall the metaphor used in [76], in which the architecture is seen as a runway for allowing fast airplanes (Agile teams) to take off and land for a mission (feature development). The RT can be appointed dynamically by a team of Product Owners and architects (see "Governance Team")

together, when a long-term refactoring is needed and therefore prioritized as more important than feature development. RTs are visible on the right in Figure 49.

10.3.2.2 Architecture Teams (ATs)

Most of the identified challenges in the architecture practices are related to a single role, but necessitate coordination and collaboration among different architects in Architecture Teams (ATs in Figure 49): for example, in *monitoring the current status of the system*, no single architects can have all the information needed: the system might have different inconsistencies with architecture at different levels (for example, low-level design on a class level or the presence of high-level dependencies among components). The complete monitoring of the system can only be achieved by continuous communication and interaction among different architects (bi-directional arrows in Figure 49): TAs belonging to different teams with a GA or GAs coordinating different groups of teams with the CA. The same degree of coordination is important for spreading the architecture, from the high level concepts expressed by the CA to the low level design implemented by the teams and known by the TA. Also when assessing the risk of architectural debt and taking decisions about solutions and changes, for example the prioritization of refactorings, the architects need to have a common forum and resources allocated for communication, analysis and tools. We found, in the organizations, the lack of an organizational mechanism such as ATs, which hindered the implementation of the practices for Governance Teams (GT).

10.3.2.3 Governance Teams (GTs)

We have shown how some of the challenges related to architecture practices that we have found can be mapped to architect roles and the Architecture Team. However, for those practices regarding "risk management" and "architecture decisions and changes", we found a strong relationship between the architects and the Product Owners or higher-level Product managers. The risk assessment of architecture changes and decisions determines the ratio of resources allocated to the improvements or of the architecture with respect to the resources used for feature development. We found the need, in the organizations, of a team involving Architects and Product Owners or Managers (Governance Teams on different levels, as illustrated in Figure 49) with the responsibility of strategically prioritizing the backlogs of the teams (dotted arrows in Figure 49) between features and architecture improvements, in order to balance the short-term with the long-term value output.

10.3.3 Overall Framework

The overall framework of roles, teams and practices. A representation is shown in Figure 49, which combines the visualization of different views: the relationships among the organizational components (architects, managers, teams) with respect to different perspectives (*Architecture* and *Governance*). Figure 49 shows also the communication needs by the architect roles (central area on the *Architecture Perspective*), between the roles and the Agile teams (left) and among the different GTs (*Governance Perspective*). Figure 49 shows the prioritization relationships among the roles and the teams (dotted arrows) and outlines, in both the perspectives, the RTs, our new concept for some of the Agile teams. The are explained in the roles and teams sections, since they could not fit in Figure 49.

10.3.4 Introduction of CAFFEA in the companies

The dynamic evaluation workshop (step 4) conducted after 6 months of introduction of CAFFEA gave us insights about the first steps and prerequisites. The most important prerequisite is for the management to invest in the organizational change. Then, the first step was to introduce TAs in the teams and to create (or redefine with proper

responsibilities) GAs. Usually a CA is already in place in the organization. The second step is to create architecture teams (ATs) including TAs, GAs and CA(s) and set up related tools and practices. Architects then were interfaced with the product management through the creation of one or more GTs (third step).

10.4 VALIDATION OF RESULTS

We cannot show all the validation data available to the researchers for space reasons, but we give an overview and we mention the most important points.

10.4.1 Validation of CAFFEA

The validation of the overall CAFFEA through the survey shows the perceived usefulness of our contribution for the 7 organizations involved in the static validation workshop (step 3). The response is very positive, with a median of 5 over 6 and a low standard deviation. Also, during the dynamic validation (after 6 months after CAFFEA introduction), some signals of better communication and cooperation among the architects and the teams have been reported. The main challenges in introducing the whole framework for the 4 interviewed companies were mainly due to challenges in "buying" management commitment and therefore receiving resource allocation for putting in place the teams and performing the practices in CAFFEA.

Table 31. Validation data for the whole framework and current position of the companies with respect to our model

Proposition	**M**	**SD**	**I**
The framework CAFFEA is sound for enhancing continuous architecting in your organization	5	0.73	4.27
Your organization can be mapped to the framework CAFFEA	4	1.24	2.76

10.4.2 Validation of teams

10.4.2.1 Governance Team

The GT has also received quite strong positive feedback (all the median values in the survey are 5), especially for the usefulness in giving strategic feedback to the FTs (such as short-term/long-term directions) and helping their coordination. Practical employment of such team has brought to the increment of the prioritization of refactoring of risky technical debt, as we could have seen by analyzing the teams and architecture teams backlogs. However, the challenges for implementing GTs were mainly socio-political: (some) Product Owners, for example in company C, were afraid of loosing resources dedicated to short-term feature development.

Table 32. Validation data for the Governance Team

Proposition	**M**	**SD**	**I**
The GT is/would be useful for providing strategic input to the XFTs	5	0.94	4.06
The GT is/would be useful for coordinating the XFTs	5	1.3	3.7
The GT should be responsible for prioritizing between short term/long term feature development	5	1.42	3.58
Within the GT , the GA should have power and resources for allocating Runway Teams	5	1.72	3.28

10.4.2.2 Architecture Team

The architecture team received a strong positive validation for implementing Agile architecting: ATs were employed in all the 4 companies that reported at the workshop after 6 months. Clear benefits were reported, such as the increment of iterative communication and cooperation among the architects and implementation of practices

for tracking and assessing the risk of (architectural) technical debt. For the AT we don't have numerical validation data, since this team was recognized after the survey.

10.4.2.3 Runway Team

According to the validation data, the (dynamic) appointment of a RT is quite well accepted as a suitable solution for supporting Agile architecting. Despite the multiple activities mentioned during the qualitative data collection as being covered by the RT, the consensus of the informants has been reached upon the need of dedicate the RT to create runway infrastructure for continuous architecture, to conduct refactorings or architecture improvements that cannot be included in feature development and to create instruments for architecture consistency checks (less support). The main challenges were related to the lack of GTs in charge of appointing RTs to such activities instead of feature development. In some cases, a fixed amount of RTs was established: such approach decreased the dynamic power of such solution, but also decreased the political frictions among the architects and product owners.

Table 33. Validation data for Runway Teams

Proposition	M	SD	I
It is necessary to appoint RT to conduct dedicated runway work	5	1.41	3.59
It is necessary to appoint a RT to improve infrastructure	6	1.56	4.44
It is necessary to appoint a RT to automate architecture consistency checks or visualization	4	1.62	2.38
It is necessary to appoint a RT to refactor the code	5	1.82	3.18
It is necessary to appoint a RT to do test activities	2	1.24	0.76
It is necessary to appoint a RT to educate other XFTs on architecture	2	1.58	0.42
It is necessary to appoint a RT to educate other XFTs on context knowledge	2	1.59	0.41
It is necessary to appoint a RT to educate other XFTs on best practices	2	1.67	0.33
It is necessary to appoint a RT to fix emergency tasks	3	1.39	1.61
The RT should be a team of high skilled employees	5	1.45	3.55

10.4.3 Validation of roles

From the validation data it's clear how all the roles (CA, GA and TA) and their mapping to the main activities were quite strongly validated by the informants. Especially the establishment of the GAs was found to be a really important improvement. However there were three main issues reported, which need further investigation:

- Could GA be *also* a TA / CA or the roles should be exclusive?
- Should the TA be connected to specific parts (e.g. components) of the system?
- Citing an informant: "*It's difficult to find a combination of specialists and strong characters*" for such role.

Table 34. Validation data for Chief Architect

Proposition	M	SD	I
CA should consult GAs and TAs in the sale phase for estimation	5	1.54	3.46
CA should collect input for high level architectural decisions through plenary sessions	5	0.75	4.25
CA should collect input for high level architectural decisions through questionnaires	3	1.18	1.82

Table 35. Validation data for the Governance Architect

Proposition	M	SD	I
GA should have power to prioritize Runway work	6	1.26	4.74
GA should be responsible for maintaining inter-feature architecture documentation	5	0.7	4.3
GA should be responsible for taking into account future development of features	5	1.36	3.64
GA should distill high level architectural patterns with the Chief Architect	5	0.68	4.32
GA should participate in FTs' retrospective sessions	5	1.15	3.85
GA should monitor architecture consistency	5	0.68	4.32

Table 36. Validation data for Team Architect

Proposition	M	SD	I
The TA is necessary in every FT	5	0.92	4.08
The TA should lift decisions about architecture evolution (collecting input from the FT)	6	0.63	5.37
The TA should investigate if the taken decisions are affecting other FTs and report to the GA	5	1.1	3.9

10.5 DISCUSSION AND CONCLUSIONS

10.5.1 Limitations and Threats to Validity

A limited amount of product owners/managers were involved, even though part of the informants had such role or have covered similar roles in the past. This calls for a further investigation on the perspective of such roles.

Given the time needed for macro-organizational changes, we could not aim at a complete validation, which is however in the researchers' long-term goal. The preliminary evaluation gives researchers and practitioners valuable insights on how CAFFEA was introduced and the challenges faced to implement CAFFEA in practice.

As for construct and internal validity, we have iteratively interviewed all the architect roles in Agile organizations, triangulating perspectives from different roles (design and architecture level), and from employees from 7 companies. As for external validity, we cannot fully generalize the results for large scale Agile software development. We limit our claims to large product line companies, and the current investigation assures good validation (6 organizations) for embedded systems development. The involvement and positive response from company E, developing pure software, suggests the application of our framework also to such domain. Conclusion validity was supported by involving 3 researchers in the investigation and by reporting workshops where employees were validating our findings [63].

10.5.2 Related work

We haven't found related work tackling the organizational aspects of employing Agile architecting in large software product line companies with ASD.

Our work takes inspiration from Leffingwell's work [76] and the concepts of architecture runway. However, the work done by Leffingwell is not supported by scientific investigation following a rigorous research process, and we employed the investigation to a specific sub-domain of large companies employing ASD.

Kettunen and Laanti [90] provide a framework, based on industrial experience, aimed at understanding how and why agility could be utilized for software process improvement (SPI) in large-scale embedded software product development. However,

the authors themselves claim to not providing organizational solutions for enabling flexibility in architecture to support Agile.

Kructhen, in [43], defines several anti-patterns for software architects, based on several experiences in architecture teams. However, the anti-patterns are not specific for a given context, which in our case was specified as large companies employing ASD and delivering product lines. We have built upon the results in [43] by using the anti-patterns for formulating the interview guide, in order to investigate the current two-ways communication practices missing in our specific context.

10.5.3 Contribution to research and practice

We have identified the main gaps in the current practices in order to employ Agile architecting in large ASD, related to the following categories:

- *architecture risk management* (prioritization of short-term and long term tasks),
- *architecture decision and changes* and
- *communication of architecture*, composed by two-way directions:
  - *Providing architecture knowledge*
  - *Monitor the current status of the system*

The combination of the previous components leads to the identification of a major gap in the current organizations, the lack of architecture technical debt management. Such phenomenon is recently being studied from different angles [33], [124], and is concerned with the organizations taking risk-informed architecture decisions about which architecture changes, such as refactoring or evolution, need to be conducted for having an acceptable ratio of cost/impact. The lack of architecture technical debt management might quickly lead the companies to crisis points where adding new business value to their products (new features or new products) incur in major efforts, paralyzing the long-term responsiveness [125].

We contribute by highlighting current challenges with respect to architectural practices (RQ1): such gaps point at the need for specific architect roles; Team architects, Chief architects and especially important is the Governance Architect, an intermediate key role for coordinating Agile architecting and scaling Agile in large organizations. Such architect roles need organizational mechanisms to cooperate, Architecture Teams, and to interface with Product Management for prioritization and decisions. We developed the CAFFEA framework, including roles, teams and practices, for giving support for Agile architecting (RQ2). Such framework, given the current identified gaps, has a specific focus on architecture technical debt management.

10.5.4 Conclusions

The short-term responsiveness in delivering value offered by ASD needs to be enhanced, in large software organizations developing embedded software, by Agile architecting, the management of a software architecture supporting long-lasting responsiveness. We have identified the gaps in the activities for conducting Agile architecting and we have developed an organizational framework, CAFFEA, including roles, teams and practices. CAFFEA has been statically validated by 40 employees from 7 different sites (including non-embedded software companies), and has also been introduced in 4 large companies. These results give guidance for the practitioners introducing ASD in large software companies (especially developing embedded software), and suggest directions for further research: future work includes additional evaluation of the benefits and challenges of CAFFEA as well as the development of practices for architecture technical debt management.

11 DISCUSSION AND CONCLUSIONS

In this Chapter we will discuss the contributions of this thesis with respect to the research questions. Then we will highlight possible future work and we will conclude with a summary of the overall contribution.

11.1 RESEARCH QUESTIONS AND CONTRIBUTIONS OF THE THESIS

11.1.1 RQ1 What factors influence long-term and short-term responsiveness?

In the first study, presented in Chapter 4, we conducted an investigation that brought to light several factors in different areas that have a positive or negative (or both) influence on short-term (*speed* in Chapter 4) and long-term responsiveness (*reuse* in Chapter 4). The main result is that organizations need to successfully manage several factors spanning different areas, technical and not. Another contribution is that particular attention is needed to be dedicated to *Interaction* factors among different actors in the software development process. Especially inter-group interactions, among teams and with other parts of the organization, need to be improved.

Such results are noteworthy in the software management field, since they show how ambidexterity seems to be a goal that is not achievable by only a few steering managers. On the contrary, in a large company such goal needs to be supported by *individual (contextual) ambidexterity* in several roles, from managers to architects to developers themselves. As discussed in the beginning of this thesis, there are several approaches aimed at achieving ambidexterity, from structural (dividing the organization in order to separately manage conflicting goals) to contextual ambidexterity. Our results support the need of contextual ambidexterity (based on individual ambidexterity) rather than structural, since several factors are intricately intertwined. Since roles have several responsibilities, it seems difficult to sharply separate who should be responsible for one side of ambidexterity only and who for the other, since the conflict would remain either within the roles or across the sub-organizations.

On the contrary, in the last Chapter we can see how there is a need to create a structure in order to manage interaction among key actors and in order to reconcile the conflicts among them. In such Chapter 10, we propose CAFFEA, an organizational solution that would create a suitable environment for managers, architects and teams. Such setup brings advantages in improving communication and in reconciling the conflicts due to different views and information owned by the different roles. In particular, the framework would have the following key activities as targets for improvement: risk management, management of decisions and changes, communication of architecture knowledge and monitoring the current status of the system.

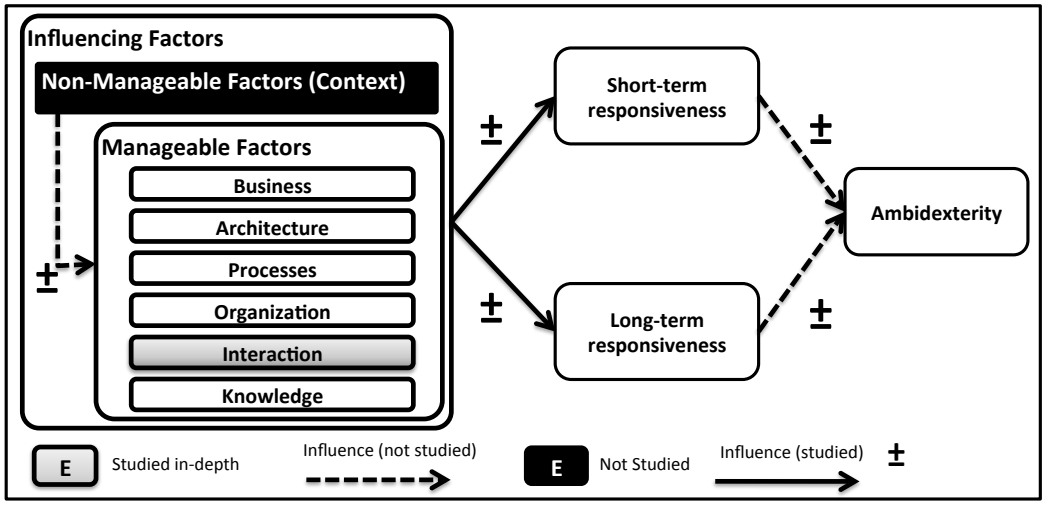


Figure 50. Interaction challenges are among the manageable factors influencing short-term and long-term responsiveness, which in turn affect ambidexterity.

11.1.2 RQ2 What interaction challenges affect ambidexterity?

Such research question can be split in two different sub-questions, depending on which perspective we take. RQ2 becomes therefore:

RQ2.1 What interaction challenges among Agile development teams affect the achievement of short-term and long-term responsiveness?

RQ2.2 What interaction challenges between Agile teams and other parts of the organization affect the achievement of short-term and long-term responsiveness?

Two chapters have been dedicated to answering this RQ. In particular, RQ2.1 is answered in Chapter 5 and RQ2.2 in Chapter 6.

As for RQ2.1, we found that there was waste caused by interactions (especially in terms of development speed and therefore responsiveness) among Agile teams. The waste was created by eight main phenomena or else called *effects*, (for example, waiting time for information) which were caused by ten factors (for example, unexpected dependencies among the features). Chapter 5 also highlights seven recommendations suggested by the practitioners in order to complement Agile practices and improve inter-team interaction.

These results are useful for the Agile teams and project managers in order to improve ambidexterity on a team level. Chapter 5 aims at improving both short-term and long-term responsiveness by improving interaction among the Agile teams. By recognizing the challenges, observable through the effects, it's possible to track them back to the root factors, which are manageable by applying the recommendations. This way, the waste can be decreased and the teams are able to deliver features faster both in the short and long term. However, these results don't take in consideration a broader perspective, considering the parts of the organization that are not only development teams. This perspective is taken when answering RQ2.2.

As for RQ2.2 we evaluated, in Chapter 6, the presence of 23 interaction challenges between the team and other groups in the organization. We quantified the presence of such challenges in three studied companies thanks to the results obtained from a survey. The results showed how all the challenges were recognized and most of them were strongly perceived as hindering short-term and long-term responsiveness (decreasing ambidexterity).

On an academic point of view, these results greatly strengthened the exploratory results obtained in Chapter 4 with new confirmatory evidences from 38 respondents in three large companies. On the practical side, the results provide a prioritized list of the main challenges across the boundaries between the Agile development team and the other groups of large organizations developing embedded software. A map of such boundaries is shown in Figure 3.

Furthermore, in Chapter 6 some of the challenges have been compared with similar studies on Agile projects that are not related to embedded software development. The results in the paper are novel since we found that some challenges were not highlighted earlier with respect to the studied domain: one of the main contextual factor was related to the need of attention, in Agile practices, to maintaining a good quality for the system and software architecture. Architecture represents a synchronization mechanism for the teams, which share the responsibility of reaching several internal and external qualities of embedded systems: among these we found reusability, which is critical to achieve long-term responsiveness, and performance, which usually represents and important customer-related quality. Another two contextual factors are related to the process aspect: the overall product management process and the disciplines not related to software engineering (such as electrical or mechanical engineering) have a different release cycle and different milestones with respect to the Agile ones, and therefore need to be synchronized.

These results are important for the theories underpinning Agile Software Development, since they highlight how the implementation of Agile practices depends on the domain on which they are applied. Clearly, for embedded software, there are interaction challenges that need to be mitigated, especially through spanning activities improving architecture management and processes.

11.1.3 RQ3 What spanning activities are needed in order to mitigate the interaction challenges affecting ambidexterity?

As mentioned for RQ2, we investigated in Chapter 5 and 6 what interaction challenges were hindering ambidexterity. In the same study (Chapter 6), we also investigated what *spanning activities* were needed in order to mitigate such challenges. One of the most occurring challenges, in the studied organizations, involved teams and (system) architects. The activities that were reported as missing by the practitioners were related to architecture management. Therefore, it was clear that spanning activities concerning *system and software architecture management* were needed.

Based on the previous contribution and employing new case-studies, in Chapter 10 we developed an organization solution, included in the overall CAFFEA framework in order to improve the interaction challenges: such solution encompasses a set of spanning activities that were missing at the studied companies and that CAFFEA is aimed at covering. Such spanning activities involve architects and developers, but we found how product managers are also critical roles both for conducting architecture management but also in consuming the information related to the status of architecture. The needed spanning activities should cover the following areas:

- *Risk management:*
  The prioritization of short-term and long-term responsiveness involves the need of understanding the risk of not being able to deliver in the present or in the future.
- *Managing decisions and changes*
  There is a need for roles responsible for inter-feature architecting and for assuring communication and cooperation between the architects and Agile teams.
- *Providing Architecture Knowledge (to the Agile teams)*

It is important to assist the teams in maintaining the architecture, both with face-to-face communication but also supported by documentation when the architecture knowledge is complex and extensive.

- *Monitoring the current status of the system*
  Often the implementation in the code quickly drifted away from the architecture defined and used for strategic decisions and risk management by architects and product managers. This phenomenon is called Architectural Technical Debt, and monitoring its accumulation in the system is critical in order to estimate risks and for architecture decision and changes (two activities mentioned earlier).

While in the previous results we focused into understanding which groups were experiencing interaction challenges hindering ambidexterity, these four areas represent the *target* for improvement. This means that the spanning activities to be implemented need to cover these areas.

In particular, these results imply that managing ATD would contribute to improve all the mentioned areas, which in turn would mitigate the interaction challenges. Therefore, a main spanning activity to be introduced is Architecture Technical Debt Management, which needs to involve architects, product managers and teams in order to balance short-term and long-term responsiveness. This represents an important implication for the Agile community: in large embedded software companies, in order to balance short-term with long-term responsiveness, managers and software improvers need to take in consideration a way to introduce practices that complement the Agile ones in order to manage architecture with a risk management perspective. Such hypothesis is also in line with a recent study that consider architecture management as a risk management activity [147]. These results also contribute to understanding *what* to improve, e.g. they offer a clear target rather than a generic and fuzzy need, as reported in other related work (see [6]).

Thanks to these analytical results, it's possible to design concrete solutions in order to mitigate interaction challenges hindering ambidexterity. Consequently, these results led to the design of the two solutions presented in the next sections: Architectural Technical Debt Management and the CAFFEA framework.

11.1.4 RQ4 What strategic information about Architecture Technical Debt needs to be shared between architects, product owners and teams in order to manage ambidexterity?

As mentioned in the previous section, ATD Management was found important in order to manage ambidexterity. This research question aimed at understanding what strategic information is needed by the stakeholders in order to perform the spanning activity related to ATD. In order to provide answers to such RQ, we divided it into three sub-questions and we performed two parallel, longitudinal studies in order to understand:

RQ4.1 What causes the accumulation of Architecture Technical Debt and its interest?

RQ4.2 How does the interest of Architecture Technical Debt affect long-term responsiveness?

RQ4.3 What refactoring strategies can be applied in order to achieve a convenient trade-off between short-term and long-term responsiveness?

We contribute to RQ4.1 with a taxonomy of the causes for the accumulation over time of ATD with respect to the different phases of development (Chapter 7). These causes are important since they need to be avoided, and therefore they have to be recognized by the stakeholders. In the ATD management activity, the practitioners will therefore need to analyze the current situation and monitor if the factors are occurring.

The main reason why the practitioners need to avoid such factors and ATD to accumulate is that such accumulation leads often to dangerous phenomena such as development crises. This model, one of the main novel results highlighted in Chapter 7, is particularly important for the strategic management of ATD with respect to ambidexterity, since it shows how ATD is continuously and unavoidably accumulated, and software will eventually contain ATD that would lead to a crisis (hindering long-term responsiveness). However, such crisis needs to be avoided or postponed as much as possible: Chapter 7 shows how it's necessary to trade some short-term responsiveness for conducting refactoring in favor of long-term responsiveness. This way, although we argue that total refactoring is not possible, partial refactoring is needed anyway in order to either minimize the number of crises or to retire the product at the end of its lifecycle before the crisis. This answers RQ4.3.

However, once understood that ATD needs to be partially refactored in order to improve long-term responsiveness, it is important to comprehend how to trade short-term responsiveness in order to achieve long-term responsiveness. The main answer is in prioritizing only those ATD items that have major negative effects on long-term responsiveness. This is summarized in RQ4.2. In order to understand what ATD needs to be refactored, we investigate more in depth what was the risk of lack of architectural quality in particularly dangerous cases. Such information is needed by architects and product managers (and, when possible, by the teams) in order to prioritize the refactoring of specific cases of lack of architectural quality, which will cause an expensive lack of long-term responsiveness.

With respect to RQ4.2, in Chapter 8 we provide a taxonomy of the classes of ATD that led, in the studied companies, to expensive long-term delays and lack of responsiveness. Another contribution is the identification of several socio-technical anti-patterns that lead to such long-term issues: three Vicious Circles and a specific phenomenon that needs to be taken in consideration: Contagious Debt. Such anti-patterns need to be recognized and stopped before they lead the ATD to be so expensive to repay that it is not possible to be fixed, and therefore a development crisis is inevitable. The information provided in Chapter 8 can therefore be used as shown in Figure 9, where a spanning object (for example, a tool providing such information), used in the ATD management spanning activity, is shared between product managers, architects and teams on the status of the system with respect to the dangerous ATD items.

To summarize the contribution, by understanding the causes and the effects of ATD (RQ4.1 and RQ4.2), we provide the key information required in the spanning activity by the stakeholders. With this information, the stakeholders can proactively prevent or monitor ATD, and can apply refactoring strategies (RQ4.3) in order to refactor the ATD items that are more dangerous for long-term responsiveness. The refactoring decisions, outcome of the ATD management activity, will then lead to balance the two kinds of responsiveness and therefore would lead to managing ambidexterity (RQ4). Such information can be also exploited either for developing methods and tools that would visualize what the stakeholders need during the spanning activity.

11.1.4.1 Evaluation of the strategic information

The strategic information described above and contributing to RQ4 (by answering the sub-questions RQ4.1-3) needed to be evaluated. In fact, we needed to collect confirmatory evidence that such information would be useful for the stakeholders when applying a spanning activity to achieve ambidexterity. In Chapter 9 we evaluated such information. The main result of this Chapter is that the research done so far has brought to light information that is useful and usable in practice by the stakeholders. The study also highlights an important result, or else that the stakeholder would employ 10-20% of the development time to manage ATD, which represents the suggested trade-off

between allocating resources to achieve short-term delivery and the ones (10-20%) dedicated to long-term responsiveness. This confirms that the studied information is critical for the companies and might have an impact on the organizations' processes to achieve ambidexterity.

11.1.5 RQ5 What organizational solution can be applied in order to facilitate spanning activities to manage ambidexterity?

In Chapter 10 we explored a possible organizational solution, CAFFEA, which would support the implementation of various spanning activities among architects, product managers and development teams. We mentioned before, in relation to RQ3, how these spanning activities include risk management, management of decisions and changes, communication of architecture knowledge and monitoring the current status of the system, and we mentioned how Architectural Technical Debt management contributes to manage the other ones as well. It is important that the collection and assessment of the ATD information would be included in the companies' processes, in order to prioritize and align short-term and long-term responsiveness. To support this, the stakeholders need an organizational structure that support this spanning activity, otherwise such activity would never occur in practice. In Chapter 10 we report an organizational solution developed on the basis of a coordination theoretical framework and several empirical experiences from 7 companies. Such solution has also been employed in practice by some of the studied companies, and we have started the evaluation of CAFFEA, providing a static evaluation (which means that the companies recognize the solution as fitting their needs without the actual implementation) and a preliminary dynamic evaluation, which shows how in practice such solution brings benefits with respect to the studied interaction challenges and help improving ambidexterity. The dynamic evaluation, which usually spans a time interval that is not controllable for the researchers, is still in progress and is not included in this thesis.

11.2 FUTURE WORK

11.2.1 Implementation of Architectural Technical Debt methods and tools

Chapters 7, 8 and 9 outline a series of results that can be used in order to define the information that needs to be retrieved and analyzed during a spanning activity among architects, developers and (in some cases) product managers in order to manage ambidexterity. Such information needs to be practically retrieved from several sources: source code, other artifacts (for example, architecture artifacts or project roadmaps), or else it has to be (semi-)manually encoded when in form of tacit knowledge. In order to facilitate such retrieval, we aim at developing a method and a tool that would assist developers, architects and product managers in carrying out some spanning activity. For example, as shown in picture Figure 9, we aim at creating an Architectural Technical Debt map of the items present in the system and their risk with respect to short-term and long-term responsiveness. This study is currently in progress.

11.2.2 Evaluation of CAFFEA (in progress)

The organizational solution included in the overall framework CAFFEA needs to be evaluated by its application in practice. Since employing an organizational framework is a task that is difficult and requires time and resources to be implemented by the companies, the evaluation is not concluded yet. However, such process is in progress in some of the companies participating in this research project and in one case we have collected data (not yet published) that show how such company is beneficing from the implementation of CAFFEA and from the related spanning activities.

11.3 CONCLUSION

Ambidexterity is a complex goal for software companies to achieve: in particular, large organizations struggle in balancing short-term and long-term responsiveness across a broad number of factors and actors, from managers to architects to developers. Current literature provides little understanding of the phenomena and only a narrow range of solutions have been proposed, mostly on a principle level and without empirical evidence especially related to the field of software engineering.

In this thesis we have enriched the software engineering body of knowledge by reporting novel results with respect to the achievement of ambidexterity. We have studied the phenomenon in depth, and we have provided a practical solution that makes possible to interact, for several stakeholders in large embedded software companies, in order to balance short-term and long-term responsiveness.

First, we have explained the complexity of the phenomenon by providing a list of many factors influencing short-term and long-term responsiveness. We show how, in order to manage ambidexterity, it is necessary to intervene on several aspects of software development, technical and not.

We have investigated several of these factors influencing ambidexterity, and we have focused the research effort in presenting and quantifying the most influential interaction challenges present in Agile organizations developing embedded software. Such challenges involve product management and architecture management and the main challenges to be mitigated are among developers, architects and product managers. We have identified Architectural Technical Debt (ATD) management as the main spanning activity that is needed in order to mitigate the challenges and enabling the active balance of short-term and long-term responsiveness.

With respect to ATD, we have studied what strategic information is needed by the stakeholder in order to manage ambidexterity. First, we provide a taxonomy of the classes of dangerous ATD items that lead to pay high interest with respect to long-term responsiveness: such information can be used for identification and prioritization purposes. Secondly, we highlight several causes of ATD accumulation together with vicious circles and the dangerous contagious debt phenomenon, which triggers the continuous growth of interest over time: such continuous growth of interest might lead to severe events such as development crises, which could stop development and therefore have a negative effect on responsiveness. Practitioners now have an instrument to proactively manage the accumulation of ATD.

The information provided in this thesis can be collected using tools or methods during a spanning activity dedicated to ATD management in software companies, in order to monitor the accumulation of dangerous ATD and to drive the decisions of employing resources to refactoring ATD. According to our results, the best strategy for refactoring is to partial refactor: it's necessary to trade part of short-term responsiveness (in the order of 10-20% of R&D investment, as suggested by the practitioners) in order to avoid the long-term crises.

Finally, we provide an organizational solution, CAFFEA, which is currently being empirically evaluated in order to allow the stakeholder to practically employ the spanning activity of managing ATD. Such solution has been found suitable by the employees at the studied organizations, and its application is providing benefits to the companies where was employed.

The results have been obtained through several phases of data collection. We have followed a thorough research process based on Grounded Theory, complying with the best principles and practices of case-study research for Software Engineering in order to provide reliable results based on the collection of a large number of evidence in collaboration with several experienced practitioners in 7 large software companies.

BIBLIOGRAPHY

[1] A. Martini, L. Pareto, and J. Bosch, "Enablers and inhibitors for speed with reuse," in *Proceedings of the 16th International Software Product Line Conference - Volume 1*, New York, NY, USA, 2012, pp. 116–125.

[2] A. Martini, L. Pareto, and J. Bosch, "Improving Businesses Success by Managing Interactions among Agile Teams in Large Organizations," in *Software Business. From Physical Products to Software Services and Solutions*, G. Herzwurm and T. Margaria, Eds. Springer Berlin Heidelberg, 2013, pp. 60–72.

[3] A. Martini, J. Bosch, and M. Chaudron, "Investigating Architectural Technical Debt Accumulation and Refactoring over Time: a Multiple-Case Study," *Inf. Softw. Technol.*

[4] A. Martini and J. Bosch, "Towards prioritizing Architecture Technical Debt: information needs of architects and product owners," presented at the 41th Euromicro SEAA conference, Funchal, Madeira.

[5] A. Martini and J. Bosch, "Towards introducing Agile Architecting in Large Companies: the CAFFEA framework," in *XP Conference 2015*.

[6] P. S. Adler, B. Goldoftas, and D. I. Levine, "Flexibility Versus Efficiency? A Case Study of Model Changeovers in the Toyota Production System," *Organ. Sci.*, vol. 10, no. 1, pp. 43–68, Feb. 1999.

[7] N. P. Napier, L. Mathiassen, and D. Robey, "Building contextual ambidexterity in a software company to improve firm-level coordination," *Eur. J. Inf. Syst.*, vol. 20, no. 6, pp. 674–690, 2011.

[8] B. Boehm, "Get ready for agile methods, with care," *Computer*, vol. 35, no. 1, pp. 64–69, Jan.

[9] B. D. Reyck, Y. Grushka-Cockayne, M. Lockett, S. R. Calderini, M. Moura, and A. Sloper, "The impact of project portfolio management on information technology projects," *Int. J. Proj. Manag.*, vol. 23, no. 7, pp. 524–537, Oct. 2005.

[10] Matthias Holweg, "The three dimensions of responsiveness," *Int. J. Oper. Prod. Manag.*, vol. 25, no. 7, pp. 603–622, Jul. 2005.

[11] 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.

[12] C. B. Gibson and J. Birkinshaw, "The Antecedents, Consequences, and Mediating Role of Organizational Ambidexterity," *Acad. Manage. J.*, vol. 47, no. 2, pp. 209–226, Apr. 2004.

[13] S. Raisch and J. Birkinshaw, "Organizational Ambidexterity: Antecedents, Outcomes, and Moderators," *J. Manag.*, vol. 34, no. 3, pp. 375–409, Jun. 2008.

[14] F. J. Linden, K. Schmid, and E. Rommes, *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer Science & Business Media, 2007.

[15] J. Bosch and P. M. Bosch-Sijtsema, "Introducing agile customer-centered development in a legacy software product line," *Softw. Pract. Exp.*, vol. 41, no. 8, pp. 871–882, 2011.

[16] R. Baskerville, J. Pries-Heje, and S. Madsen, "Post-agility: What follows a decade of agility?," *Inf. Softw. Technol.*, vol. 53, no. 5, pp. 543–555, May 2011.

[17] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. Martin, S. Mallor, K. Shwaber, and J. Sutherland, "The Agile Manifesto," 2001.

[18] P. Abrahamsson, J. Warsta, M. T. Siponen, and J. Ronkainen, "New directions on agile methods: a comparative analysis," in *25th International Conference on Software Engineering, 2003. Proceedings*, 2003, pp. 244–254.

[19] T. Dybå and T. Dingsøyr, "Empirical studies of agile software development: A systematic review," *Inf. Softw. Technol.*, vol. 50, no. 9–10, pp. 833–859, Aug. 2008.

[20] K. Beck, *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2000.

[21] M. Lindvall, D. Muthig, A. Dagnino, C. Wallin, M. Stupperich, D. Kiefer, J. May, and T. Kahkonen, "Agile software development in large organizations," *Computer*, vol. 37, no. 12, pp. 26 – 34, Dec. 2004.

[22] X. Wang, K. Conboy, and O. Cawley, "'Leagile' software development: An experience report analysis of the application of lean approaches in agile software development," *J. Syst. Softw.*, vol. 85, no. 6, pp. 1287–1299, Jun. 2012.

[23] T. Dingsoyr, T. Dyba, and P. Abrahamsson, "A Preliminary Roadmap for Empirical Research on Agile Software Development," in *Agile, 2008. AGILE '08. Conference*, 2008, pp. 83–94.

[24] 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.

[25] M. Xie, M. Shen, G. Rong, and D. Shao, "Empirical studies of embedded software development using agile methods: a systematic review," in *Proceedings of the 2nd international workshop on Evidential assessment of software technologies*, 2012, pp. 21–26.

[26] J. Ronkainen and P. Abrahamsson, "Software Development under Stringent Hardware Constraints: Do Agile Methods Have a Chance?," in *Extreme Programming and Agile Processes in Software Engineering*, vol. 2675, M. Marchesi and G. Succi, Eds. Springer Berlin / Heidelberg, 2003, pp. 1012–1012.

[27] D. E. Strode, S. L. Huff, B. Hope, and S. Link, "Coordination in co-located agile software development projects," *J. Syst. Softw.*, vol. 85, no. 6, pp. 1222–1238, Giugno 2012.

[28] L. Pareto, A. B. Sandberg, P. Eriksson, and S. Ehnebom, "Collaborative prioritization of architectural concerns," *J. Syst. Softw.*, vol. 85, no. 9, pp. 1971–1994, Sep. 2012.

[29] N. B. Moe, A. Aurum, and T. Dybå, "Challenges of shared decision-making: A multiple case study of agile software development," *Inf. Softw. Technol.*, vol. 54, no. 8, pp. 853–865, Aug. 2012.

[30] C. Lassenius, T. Dingsøyr, and M. Paasivaara, Eds., *Management Ambidexterity: A Clue for Maturing in Agile Software Development*, vol. 212. Cham: Springer International Publishing, 2015.

[31] 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.

[32] L. Cao and B. Ramesh, "Agile Requirements Engineering Practices: An Empirical Study," *IEEE Softw.*, vol. 25, no. 1, pp. 60–67, Jan. 2008.

[33] D. St\a ahl and J. Bosch, "Modeling Continuous Integration Practice Differences in Industry Software Development," *J Syst Softw*, vol. 87, pp. 48–59, Jan. 2014.

[34] M. Shaw and P. Clements, “The golden age of software architecture,” *Softw. IEEE*, vol. 23, no. 2, pp. 31–39, 2006.

[35] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[36] Z. Li, P. Avgeriou, and P. Liang, “A systematic mapping study on technical debt and its management,” *J. Syst. Softw.*, vol. 101, pp. 193–220, Mar. 2015.

[37] W. Cunningham, “The WyCash portfolio management system,” in *ACM SIGPLAN OOPS Messenger*, 1992, vol. 4, pp. 29–30.

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

[39] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, and others, “Managing technical debt in software-reliant systems,” in *Proceedings of the FSE/SDP workshop on Future of software engineering research*, 2010, pp. 47–52.

[40] 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.

[41] 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.

[42] J.-L. Letouzey, “The SQALE Method for Evaluating Technical Debt,” in *Proceedings of the Third International Workshop on Managing Technical Debt*, Piscataway, NJ, USA, 2012, pp. 31–36.

[43] 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.

[44] 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.

[45] K. Schmid, “A formal approach to technical debt decision making,” in *Proceedings of the 9th international ACM Sigsoft conference on Quality of software architectures*, 2013, pp. 153–162.

[46] ISO - International Organization for Standardization, “System and software quality models.” [Online]. Available: http://www.iso.org/iso/catalogue_detail.htm?csnumber=52075. [Accessed: 08-Mar-2015].

[47] A. Martini and J. Bosch, “The Danger of Architectural Technical Debt: Contagious Debt and Vicious Circles,” in *accepted for publication at WICSA 2015*, Montreal, Canada.

[48] P. Kruchten, “What do software architects really do?,” *J. Syst. Softw.*, vol. 81, no. 12, pp. 2413–2416, Dec. 2008.

[49] A. Martini, L. Pareto, and J. Bosch, “Role of Architects in Agile Organizations,” in *Continuous Software Engineering*, J. Bosch, Ed. Springer International Publishing, 2014, pp. 39–50.

[50] M. A. Babar and I. Gorton, “Comparison of scenario-based software architecture evaluation methods,” in *Software Engineering Conference, 2004. 11th Asia-Pacific*, 2004, pp. 600–607.

[51] 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.

[52] F. van der Linden, J. Bosch, E. Kamsties, K. Känsälä, and H. Obbink, “Software Product Family Evaluation,” in *Software Product Lines*, R. L. Nord, Ed. Springer Berlin Heidelberg, 2004, pp. 110–129.

[53] S. Betz and C. Wohlin, “Alignment of Business, Architecture, Process, and Organisation in a Software Development Context,” in *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, New York, NY, USA, 2012, pp. 239–242.

[54] C. Hofmeister, P. Kruchten, R. L. Nord, H. Obbink, A. Ran, and P. America, “A general model of software architecture design derived from five industrial approaches,” *J. Syst. Softw.*, vol. 80, no. 1, pp. 106–126, Jan. 2007.

[55] T. Chow and D. B. Cao, “A survey study of critical success factors in agile software projects,” *J. Syst. Softw.*, vol. 81, no. 6, pp. 961–971, 2008.

[56] F. M. Santos and K. M. Eisenhardt, “Organizational Boundaries and Theories of Organization,” *Organ. Sci.*, vol. 16, no. 5, pp. 491–508, Sep. 2005.

[57] N. Levina and E. Vaast, “The Emergence of Boundary Spanning Competence in Practice: Implications for Implementation and Use of Information Systems,” *MIS Q.*, vol. 29, no. 2, pp. 335–363, Jun. 2005.

[58] T. W. Malone and K. Crowston, “The Interdisciplinary Study of Coordination,” *ACM Comput Surv*, vol. 26, no. 1, pp. 87–119, Mar. 1994.

[59] A. Sandberg, L. Pareto, and T. Arts, “Agile Collaborative Research: Action Principles for Industry-Academia Collaboration,” *IEEE Softw.*, vol. 28, no. 4, pp. 74–83, 2011.

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

[61] “Bryant, A., & Charmaz, K. (Eds.). (2007). The Sage handbook of grounded theory. Thousand Oaks, CA: Sage.,” .

[62] B. G. Glaser and J. Holton, *Discovery of Grounded Theory*. 1967.

[63] M. Crotty, *The Foundations of Social Research: Meaning and Perspective in the Research Process*. Sage Publications, 1998.

[64] C. B. Seaman, “Qualitative methods in empirical studies of software engineering,” *IEEE Trans. Softw. Eng.*, vol. 25, no. 4, pp. 557 –572, Aug. 1999.

[65] R. Suddaby, “From the editors: What grounded theory is not,” *Acad. Manage. J.*, vol. 49, no. 4, pp. 633–642, 2006.

[66] B. G. Glaser and A. L. Strauss, *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Transaction Publishers, 2009.

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

[68] 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.

[69] V. R. Basili, R. W. Selby, and D. H. Hutchens, “Experimentation in software engineering,” *IEEE Trans. Softw. Eng.*, vol. SE-12, no. 7, pp. 733 –743, Jul. 1986.

[70] L. Dickens and K. Watkins, “Action Research: Rethinking Lewin,” *Manag. Learn.*, vol. 30, no. 2, pp. 127–140, Jun. 1999.

[71] R. Wieringa and M. Daneva, “Six strategies for generalizing software engineering theories,” *Sci. Comput. Program.*, vol. 101, pp. 136–152, Apr. 2015.

[72] J. Singer and N. Vinson, “Ethical issues in empirical studies of software engineering,” 2002.

[73] U. Flick, *An Introduction to Qualitative Research*. SAGE, 2009.

[74] R. Czaja and J. Blair, *Designing Surveys: A Guide to Decisions and Procedures*. Pine Forge Press, 2005.

[75] A. Martini, L. Pareto, and J. Bosch, "Communication factors for speed and reuse in large-scale agile software development," in *Proceedings of the 17th International Software Product Line Conference*, New York, NY, USA, 2013, pp. 42–51.

[76] J. Díaz, J. Pérez, P. P. Alarcón, and J. Garbajosa, "Agile product line engineering-a systematic literature review," *Softw. Pract. Exp.*, vol. 41, no. 8, pp. 921–941, Jul. 2011.

[77] G. K. Hanssen and T. E. F\aegri, "Process fusion: An industrial case study on agile software product line engineering," *J. Syst. Softw.*, vol. 81, no. 6, pp. 843–854, 2008.

[78] J. D. McGregor, "Agile Software Product Lines, Deconstructed." [Online]. Available: http://www.jot.fm/issues/issue_2008_11/column1/. [Accessed: 10-Sep-2015].

[79] K. Petersen and C. Wohlin, "A comparison of issues and advantages in agile and incremental development between state of the art and an industrial case," *J. Syst. Softw.*, vol. 82, no. 9, pp. 1479–1490, 2009.

[80] D. Leffingwell, *Scaling Software Agility: Best Practices for Large Enterprises*. Pearson Education, 2007.

[81] A. Dubois and L.-E. Gadde, "Systematic combining: an abductive approach to case research," *J. Bus. Res.*, vol. 55, no. 7, pp. 553–560, Jul. 2002.

[82] A. Martini, "Factors influencing reuse and speed in three organizations," 2012.

[83] A. Martini, "Codes supporting results in accumulation and refactoring of TD: https://dl.dropboxusercontent.com/u/41579684/Codes_grounded_more_than_2_q.xml." .

[84] D. Peterson, "Economics of software product lines," *Softw. Prod.-Fam. Eng.*, pp. 381–402, 2004.

[85] K. Schmid, "A quantitative model of the value of architecture in product line adoption," *Softw. Prod.-Fam. Eng.*, pp. 32–43, 2004.

[86] W. Tracz, *Confessions of a Used Program Salesman: Institutionalizing Software Reuse*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[87] J. Gaffney Jr and R. Cruickshank, "A general economics model of software reuse," in *Proceedings of the 14th international conference on Software engineering*, 1992, pp. 327–337.

[88] M. Morisio, M. Ezran, and C. Tully, "Success and failure factors in software reuse," *Softw. Eng. IEEE Trans. On*, vol. 28, no. 4, pp. 340–357, 2002.

[89] T. Menzies and J. S. Di Stefano, "More success and failure factors in software reuse," *Softw. Eng. IEEE Trans. On*, vol. 29, no. 5, pp. 474–477, 2003.

[90] A. Lynex and P. J. Layzell, "Organisational considerations for software reuse," *Ann. Softw. Eng.*, vol. 5, no. 1, pp. 105–124, 1998.

[91] G. Kakarontzas, I. Stamelos, and P. Katsaros, "Product Line Variability with Elastic Components and Test-Driven Development," 2008, pp. 146–151.

[92] Y. Ghanam, F. Maurer, P. Abrahamsson, and K. Cooper, "A report on the XP workshop on agile product line engineering," *ACM SIGSOFT Softw. Eng. Notes*, vol. 34, no. 5, p. 25, Oct. 2009.

[93] D. Turk, R. France, and B. Rumpe, "Limitations of agile software processes," in *Third International Conference on eXtreme Programming and Agile Processes in Software Engineering (XP 2002)*, 2002.

[94] P. Kettunen and M. Laanti, “Combining agile software projects and large-scale organizational agility,” *Softw. Process Improv. Pract.*, vol. 13, no. 2, pp. 183–193, Mar. 2008.

[95] E. Hossain, M. A. Babar, and H. Paik, “Using Scrum in Global Software Development: A Systematic Literature Review,” in *Fourth IEEE International Conference on Global Software Engineering, 2009. ICGSE 2009*, 2009, pp. 175 –184.

[96] J. Bosch and P. Bosch-Sijtsema, “From integration to composition: On the impact of software product lines, global development and ecosystems,” *J. Syst. Softw.*, vol. 83, no. 1, pp. 67–76, Jan. 2010.

[97] M. Pikkarainen, J. Haikara, O. Salo, P. Abrahamsson, and J. Still, “The impact of agile practices on communication in software development,” *Empir. Softw. Eng.*, vol. 13, no. 3, pp. 303–337, 2008.

[98] J. D. Herbsleb and A. Mockus, “An empirical study of speed and communication in globally distributed software development,” *IEEE Trans. Softw. Eng.*, vol. 29, no. 6, pp. 481 – 494, Jun. 2003.

[99] L. Layman, L. Williams, D. Damian, and H. Bures, “Essential communication practices for Extreme Programming in a global software development team,” *Inf. Softw. Technol.*, vol. 48, no. 9, pp. 781–794, Sep. 2006.

[100] J. Espinosa, S. Slaughter, R. Kraut, and J. Herbsleb, “Team Knowledge and Coordination in Geographically Distributed Software Development,” *J. Manag. Inf. Syst.*, vol. 24, no. 1, pp. 135–169, Jul. 2007.

[101] M. Korkala and P. Abrahamsson, “Communication in Distributed Agile Development: A Case Study,” in *33rd EUROMICRO Conference on Software Engineering and Advanced Applications, 2007*, 2007, pp. 203 –210.

[102] M. Cataldo, J. D. Herbsleb, and K. M. Carley, “Socio-technical congruence: a framework for assessing the impact of technical and work dependencies on software development productivity,” in *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, New York, NY, USA, 2008, pp. 2–11.

[103] S. J. Karau and J. R. Kelly, “The effects of time scarcity and time abundance on group performance quality and interaction process,” *J. Exp. Soc. Psychol.*, vol. 28, no. 6, pp. 542–571, Nov. 1992.

[104] D. Karlstrom and P. Runeson, “Combining agile methods with stage-gate project management,” *IEEE Softw.*, vol. 22, no. 3, pp. 43 – 49, Jun. 2005.

[105] O. Gotel, V. Kulkarni, M. Say, C. Scharff, and T. Sunetnanta, “Quality indicators on global software development projects: does ‘getting to know you’ really matter?,” *J. Softw. Evol. Process*, vol. 24, no. 2, pp. 169–184, 2012.

[106] K. S. Pawar and S. Sharifi, “Virtual collocation of design teams: coordinating for speed,” *Int. J. Agile Manag. Syst.*, vol. 2, no. 2, pp. 104–113, Aug. 2000.

[107] R. Giuffrida and Y. Dittrich, “Empirical studies on the use of social software in global software development – A systematic mapping study,” *Inf. Softw. Technol.*

[108] T. Kahkonen, “Agile methods for large organizations - building communities of practice,” in *Agile Development Conference, 2004*, 2004, pp. 2 – 10.

[109] D. H. Gobeli, H. F. Koenig, and I. Bechinger, “Managing conflict in software development teams: a multilevel analysis,” *J. Prod. Innov. Manag.*, vol. 15, no. 5, pp. 423–435, Sep. 1998.

[110] C. Loureiro-Koechlin, “A theoretical framework for a structuration model of social issues in software development in information systems,” *Syst. Res. Behav. Sci.*, vol. 25, no. 1, pp. 99–109, 2008.

[111] J. Y.-C. Liu, H.-G. Chen, C. C. Chen, and T. S. Sheu, “Relationships among interpersonal conflict, requirements uncertainty, and software project performance,” *Int. J. Proj. Manag.*, vol. 29, no. 5, pp. 547–556, Jul. 2011.

[112] A. Greve, M. Benassi, and A. D. Sti, “Exploring the contributions of human and social capital to productivity,” *Int. Rev. Sociol.*, vol. 20, no. 1, pp. 35–58, Mar. 2010.

[113] J. Saldaña-Ramos, A. Sanz-Esteban, J. García, and A. Amescua, “Skills and abilities for working in a global software development team: a competence model,” *J. Softw. Evol. Process*, p. n/a–n/a, 2013.

[114] T. Dybå and T. Dingsøyr, “Empirical studies of agile software development: A systematic review,” *Inf. Softw. Technol.*, vol. 50, no. 9–10, pp. 833–859, Aug. 2008.

[115] J. D. Blackburn, G. D. Scudder, and L. N. Van Wassenhove, “Improving speed and productivity of software development: a global survey of software developers,” *IEEE Trans. Softw. Eng.*, vol. 22, no. 12, pp. 875–885, 1996.

[116] Z. Ma, J. S. Collofello, and D. E. Smith-Daniels, “Causes and solutions for schedule slippage: a survey of software projects,” in *Performance, Computing, and Communications Conference, 2000. IPCCC ’00. Conference Proceeding of the IEEE International*, 2000, pp. 373–379.

[117] Z. Ma, J. S. Collofello, and D. E. Smith-Daniels, “Improving software on-time delivery: an investigation of project delays,” in *2000 IEEE Aerospace Conference Proceedings*, 2000, vol. 4, pp. 421–434 vol.4.

[118] M. Poppendieck and T. Poppendieck, *Implementing Lean Software Development: From Concept to Cash (The Addison-Wesley Signature Series)*. Addison-Wesley Professional, 2006.

[119] M. Poppendieck, “Lean software development,” in *Companion to the proceedings of the 29th International Conference on Software Engineering*, 2007, pp. 165–166.

[120] S. Lee and H.-S. Yong, “Distributed agile: project management in a global environment,” *Empir. Softw. Eng.*, vol. 15, no. 2, pp. 204–217, Apr. 2010.

[121] C. Wohlin, P. Runeson, M. Host, C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering: an Introduction*. Kluver Academic Publishers, 2000.

[122] E. C. Lee, “Forming to Performing: Transitioning Large-Scale Project Into Agile,” in *Agile, 2008. AGILE ’08. Conference*, 2008, pp. 106 –111.

[123] M. Paasivaara and C. Lassenius, “Collaboration practices in global inter-organizational software development projects,” *Softw. Process Improv. Pract.*, vol. 8, no. 4, pp. 183–199, 2003.

[124] 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.

[125] 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.

[126] 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 Conference on Software Architecture (WICSA) and European Conference on Software Architecture (ECSA)*, 2012, pp. 91–100.

[127] 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.

[128] 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.

[129] 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.

[130] 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.

[131] M. Fowler, “Technical Debt Quadrant,” 2009. .

[132] P. Berander and A. Andrews, “Requirements prioritization,” in *Engineering and managing software requirements*, Springer, 2005, pp. 69–94.

[133] “businessdictionary.com.” .

[134] N. Zazworka, C. Seaman, and F. Shull, “Prioritizing design debt investment opportunities,” in *Proceedings of the 2nd Workshop on Managing Technical Debt*, New York, NY, USA, 2011, pp. 39–42.

[135] T. Mens and T. Tourwe, “A survey of software refactoring,” *IEEE Trans. Softw. Eng.*, vol. 30, no. 2, pp. 126–139, 2004.

[136] J. Díaz, J. Pérez, and J. Garbajosa, “Agile Product-Line Architecting in Practice: A Case Study in Smart Grids,” *Inf. Softw. Technol.*

[137] S. Bellomo, R. L. Nord, and I. Ozkaya, “A study of enabling factors for rapid fielding combined practices to balance speed and stability,” in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 982–991.

[138] A. Tang, P. Avgeriou, A. Jansen, R. Capilla, and M. Ali Babar, “A comparative study of architecture knowledge management tools,” *J. Syst. Softw.*, vol. 83, no. 3, pp. 352–370, Mar. 2010.

[139] L. Pareto, P. Eriksson, and S. Ehnebom, “Architectural descriptions as boundary objects in system and design work,” *Model Driven Eng. Lang. Syst.*, pp. 406–419, 2010.

[140] B. J. Williams and J. C. Carver, “Characterizing software architecture changes: A systematic review,” *Inf. Softw. Technol.*, vol. 52, no. 1, pp. 31–51, Jan. 2010.

[141] L. de Silva and D. Balasubramaniam, “Controlling software architecture erosion: A survey,” *J. Syst. Softw.*, vol. 85, no. 1, pp. 132–151, Jan. 2012.

[142] A. Qumer, “Defining an Integrated Agile Governance for Large Agile Software Development Environments,” in *Agile Processes in Software Engineering and Extreme Programming*, G. Concas, E. Damiani, M. Scotto, and G. Succi, Eds. Springer Berlin Heidelberg, 2007, pp. 157–160.

[143] M. Drury, K. Conboy, and K. Power, “Obstacles to decision making in Agile software development teams,” *J. Syst. Softw.*, vol. 85, no. 6, pp. 1239–1254, Jun. 2012.

[144] O. Zimmermann, C. Miksovic, and J. M. Küster, “Reference architecture, metamodel, and modeling principles for architectural knowledge management in information technology services,” *J. Syst. Softw.*, vol. 85, no. 9, pp. 2014–2033, Sep. 2012.

[145] H. Unphon and Y. Dittrich, “Software architecture awareness in long-term software product evolution,” *J. Syst. Softw.*, vol. 83, no. 11, pp. 2211–2226, Nov. 2010.

[146] J. McAvoy and T. Butler, “The impact of the Abilene Paradox on double-loop learning in an agile team,” *Inf. Softw. Technol.*, vol. 49, no. 6, pp. 552–563, Jun. 2007.

[147] E. R. Poort and H. Van Vliet, “Architecting as a Risk- and Cost Management Discipline,” in *2011 9th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2011, pp. 2–11.