Jan Bosch *Editor*

Continuous Software Engineering

Springer

Continuous Software Engineering

Jan Bosch
Editor

Continuous Software Engineering

Springer

Editor
Jan Bosch
Chalmers University of Technology
Gothenburg, Sweden

ISBN 978-3-319-11282-4 ISBN 978-3-319-11283-1 (eBook)
DOI 10.1007/978-3-319-11283-1
Springer Cham Heidelberg New York Dordrecht London

Library of Congress Control Number: 2014956014

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

This book is dedicated to

Lars Pareto

who unexpectedly passed away in 2013.

We miss your passion,
your creativity and
your dedication to research,
but above all we miss you
as a friend and colleague.

Foreword

Engineering complex software systems is a true engineering challenge mostly based on human-based approaches! Transferring leading-edge software engineering approaches into practice is a challenging task and requires close—laboratory-style—collaboration between research and practice.

This thesis is based on frequent lessons learned in the past, where innovative software engineering approaches were introduced in practice without close collaboration with research and did not produce sustainable improvements. What had happened? New development approaches were introduced without measuring their effects, adapting them to specific company needs, and without continuously improving them. As a result, software developers were not convinced about the benefits for their work and tended to fall back to previous practice. In that sense, the investment into new development approaches did not show any return on investment.

Best practice examples for appropriate technology transfer—based on close collaboration, measurement of effects, and continuous improvement—are the Software Engineering Laboratory (SEL) at NASA's Goddard Space Flight Center in the USA under the leadership of Victor Basili, Frank McGarry, and Jerry Page and the Fraunhofer Institute for Experimental Software Engineering (IESE) in Germany under the leadership of Dieter Rombach and Peter Liggesmeyer.

NASA's SEL was a close collaboration between the University of Maryland (research), NASA's Goddard Space flight Center (owner of the satellite control software systems), and Computer Science Corporation (software contractor to NASA). In close collaboration, strengths and weaknesses of development practices were analyzed quantitatively, and new development approaches were prepared by research and introduced by means of a controlled technology transfer process (accompanied by controlled experiments and case studies). As a result of this approach, innovative approaches such as formal reviews, Cleanroom development, and systematic reuse were introduced sustainably, and as a result the KPIs in terms of quality, effort, and time were improved by orders of magnitude over a number of years. The SEL can be considered the "mother" of all research and technology

transfer organizations based on close research-practice collaboration. The SEL received the first International Process Improvement award from IEEE and the Software Engineering Institute at Carnegie Mellon University.

Fraunhofer's IESE has established close collaborations with companies from all sectors of industry in Germany, Europe, and beyond. Its competence is in software and system engineering. It is considered a leader in applied research and technology transfer related to scalable software engineering approaches, guaranteeing certain qualities, and being applicable for all software-enabled innovations. Most of the its customers are companies (large, medium, and small) from embedded system domains (e.g., automotive, aerospace, medical devices), software and information system domains (e.g., banking), or combinations of both (e.g., so-called smart ecosystems in the areas of mobility, health, and energy management). Companies receive sustained improvements of their software and system development capabilities as well as ideas and concepts for new product ideas and business models. Fraunhofer IESE is known foremost for its technologically sound and practically applicable approaches for requirements engineering, architecture and software product lines, automated testing, safety and security analysis and engineering, and user experience generation.

The **Software Center** presented in this book is another remarkable organization aimed at excellent applied research and technology transfer based on close collaboration between research and practice. The specific focus of the Software Center is on **continuous deployment of software**.

The traditional process-based software development based on life cycle phases with well-defined milestones has been challenged by so-called agile development approaches aiming at development time reduction without sacrificing the resulting product quality. We have learned as a community that agile development approaches cannot replace process-based development approaches as a whole. Instead we have learned that depending on application domain, criticality, size, and qualification of people, either model may be the most appropriate. This has been a revolution in that people began to understand that there is no silver bullet process model, but the process model is a variable. Since then technology advances such as Web 2.0 or SaaS have required a significant increase of releases in order to optimize customer benefits. The Software Center explores the requirements and processes most beneficial for such contexts. I am convinced, we have learned and will continue to learn that—similar to the situation when agile complemented process-based approaches—there will continue to be a justification for each approach, depending on objectives and project context.

I expect the Software Center to continue to successfully complement other existing research and technology transfer centers such as Fraunhofer IESE with a specific focus on software development in the context of Web 2.0 and SaaS. This book is an excellent introduction into the principles and works of the Software Center. I wish the organizers of the center continued success not only for their own sake but also for the sake of the European software development industry.

Kaiserslautern, Germany
July 2014

Dieter Rombach

Preface

As the rate of change and risk of disruption increase relentlessly, companies are constantly battling to proactively adopt new innovations, be it business, technology, or process innovations. No field is more intensely subject to this than the software-intensive systems industry. Ranging from automotive, defense, and telecommunications systems to large, complex installed software solutions, the companies in this industry have been subject to business model innovations, e.g., the transition from products to services; to technology innovations, such as cloud computing and real-time connectivity; and to process innovations, such as agile development practices, continuous integration, and continuous deployment. The challenge for software-intensive systems companies is how to maintain or even improve their competitive position while responding to these disruptions to the normal way of doing things.

Similarly, software engineering research is experiencing its own set of forces in that during the last decades, very few major, industry-changing new innovations have originated in academia. Instead, industry has taken over the role of introducing and driving large-scale adoption of new innovations. For instance, a business model innovation such as open-source software originated in industry. Similarly, technology innovations such as programming languages, ranging from Java to Scala, as well as integrated development environments, such as Eclipse, find their roots in industry. And finally, process innovations such as agile development and continuous integration originate in industry, rather than in academia.

Universities are the homes of numerous highly intelligent, well-trained, and well-intended individuals that are committed to making an impact, and software engineering research groups and departments are no exception. What can then be the reason for the lack of major innovations originating from academia? There are, I believe, three main reasons: First, for a variety of reasons, discussed below, software engineering researchers often have difficulty to gain access to their research environment, which are the large-scale software R&D organizations where software engineering happens. As a consequence, researchers instead focus on small-scale problems that can be studied in a university context, such as studying student projects or otherwise studying simulated, rather than real, environments. Second,

especially over the last decade, the software engineering research community has increasingly demanded empirical data to back up any research claims. This had the intention of reducing the amount of "advocacy research," i.e., researchers presenting claims and providing logically sounding arguments why these claims could be assumed to be true but without any real evidence that the intended outcomes would be seen in reality as well. Although the demand for data accomplished the intended effect, there was an additional effect: software engineering researchers increasingly studied and reported on the current state at software companies as this was the only way to collect relevant data. However, they were no longer innovating on how to improve the current state of practice as the results would not be publishable anyway. Instead, this task increasingly fell to industry. Third, and perhaps most important, academic researchers are not exposed to the market forces experienced by software-intensive systems industries and consequently focus their efforts predominantly on adding more detail, more steps, more activities, more documentation, more intermediate artifacts, more specialization of roles, etc. This focus runs counter on the pressures experienced by industry where the focus is on translating identified customer needs to solutions in the hands of customers as rapidly as possible with as little detail, as few steps and activities, as little or no documentation, and as few artifacts except code as possible, preferably accomplished by anyone who is available for the task at hand. This easily causes a certain level of arrogance among software engineering researchers and a belittling of the accomplishments of numerous outstanding engineers in industry as the goals that these engineers are, consciously or unconsciously, working towards are not properly understood by researchers who project their own goals on industrial practice.

As one may understand from the above, it has proven to be notoriously difficult to build effective, scalable, and long-term software engineering research collaborations between industry and academia. Of course, there are many examples of individual researchers or small groups collaborating for years with a company. And there are examples of companies that have gone out of their way to build relationships with researchers that have lasted for extended periods of time. However, examples of collaborations between sizable groups of relatively diverse software engineering researchers and groups of companies with similar challenges are few and far between. In fact, one of the few long-standing examples of a collaboration of this type is the Fraunhofer Institute for Experimental Software Engineering, and consequently, I am grateful that Professor Dieter Rombach has graciously agreed to provide a foreword for this book.

It was with this understanding of the challenges of collaboration between industry and academia in the area of software engineering that we started the Software Center in 2011. Initially the collaboration started with four founding companies, i.e., Ericsson, AB Volvo, Volvo Car Corporation, and Saab Electronic Defense Systems, and the combined software engineering division between Chalmers University of Technology and Gothenburg University, with three projects and a handful of researchers. Three years later, at the time of writing, we have eight companies and three universities, 15 research projects, and dozens of researchers involved in the Software Center.

Based on the above, it's clear we are on to something. So, what are the mechanisms that have made Software Center successful? There are at least three

basic principles that are worth sharing here: First, all research takes place in 6-month sprints. A sprint starts in January or July and runs for 6 months. During a sprint, the project goes through a full cycle of defining the research problem, designing the research, collecting data or conducting the experiment or trial, analyzing the results and presenting the results to the companies involved, as well as publishing the research outcomes. Each project has a long-term goal and runs for multiple or many sprints, but every sprint results relevant to the companies have to be presented. Second, the technical experts at the companies decide what research is conducted. At the end of every sprint, each ongoing project, as well as each newly proposed project, presents a plan for what to study next. A task force consisting of technical experts at the Software Center companies decides on a ranking of research projects and potentially "kills" projects that are not delivering results relevant to the member companies. This puts an equal balance on academic excellence and industrial relevance. Finally, the longitudinal nature of projects allows researchers to study current state at the member companies, but subsequently to propose improvements. If the improvements are sufficiently appealing, some or all of the software center companies will experiment with the improvement and, if successful, deploy it broadly in the respective companies. This allows software engineering researchers to be involved in and report on improvements in the way software engineering is conducted in world-class companies. The advantage to researchers, obviously, is that it is possible to study more than "current state" as well as the ability to validate innovations at multiple companies, increasing the validity as well as the ease of publication of research conducted in the scope of the Software Center.

Concluding, Software Center is an experiment to establish an effective, scalable, and long-term software engineering research collaboration between academia and industry. The book that you're holding presents the results from the first 3 years. The experiment, so far, is successful in that more companies, universities, and researchers are joining the initiative. Also, many of the results, including the Stairway to Heaven model, the CAFFEA model, the CIViT model, the HYPEX model, as well as many other results, have been adopted or are in the process of being adopted by the partner companies. Finally, over the last 3 years, the partner companies have progressed from experimenting with agile work practices to broad deployment of continuous integration in an agile teams context and the experimentation with continuous deployment of software with selected customers for some companies. As Software Center, our goal is to help companies change faster than without our involvement, and the evidence to date is that we're delivering on that goal.

This book presents the results of the first phase of the Software Center, but it also celebrates the great progress accomplished at the partner companies due to the tireless efforts of the researchers in the Software Center and the champions at partner companies. As director, I am humbled and grateful to everyone involved. All have stepped up to the challenge and actively collaborated to create something that is so much more than the sum of its parts.

Gothenburg, Sweden Jan Bosch
June 2014

Contents

Part I Introduction

1 Continuous Software Engineering: An Introduction 3
Jan Bosch

2 Climbing the "Stairway to Heaven": Evolving From Agile Development to Continuous Deployment of Software 15
Helena Holmström Olsson and Jan Bosch

3 Academia–Industry Collaboration: Getting Closer is the Key! 29
Anna Sandberg

Part II Agile Practices

4 Role of Architects in Agile Organizations 39
Antonio Martini, Lars Pareto, and Jan Bosch

5 Teams Interactions Hindering Short-Term and Long-Term Business Goals 51
Antonio Martini, Lars Pareto, and Jan Bosch

6 A Framework for Speeding Up Interactions Between Agile Teams and Other Parts of the Organization 67
Antonio Martini, Lars Pareto, and Jan Bosch

7 Customer-Specific Teams for Agile Evolution of Large-Scale Embedded Systems 83
Helena Holmström Olsson, Anna B. Sandberg, and Jan Bosch

Part III Continuous Integration

8 The CIViT Model in a Nutshell: Visualizing Testing Activities to Support Continuous Integration 97
Agneta Nilsson, Jan Bosch, and Christian Berger

9 Continuous Integration Flows 107
Daniel Ståhl and Jan Bosch

10 Towards Continuous Integration for Cyber-Physical Systems on the Example of Self-Driving Miniature Cars 117
Christian Berger

11 Industrial Application of Visual GUI Testing: Lessons Learned . . . 127
Emil Alégroth and Robert Feldt

Part IV R&D as an Innovation System

12 Post-deployment Data Collection in Software-Intensive Embedded Products 143
Helena Holmström Olsson and Jan Bosch

13 The HYPEX Model: From Opinions to Data-Driven Software Development 155
Helena Holmström Olsson and Jan Bosch

Part V Organizational Performance Metrics

14 Profiling Prerelease Software Product and Organizational Performance 167
Vard Antinyan, Miroslaw Staron, and Wilhelm Meding

15 Industrial Self-Healing Measurement Systems 183
Miroslaw Staron and Wilhelm Meding

Part VI Industry Best Practices and Case Studies

16 Experiences from Implementing Agile Ways of Working in Large-Scale System Development 203
Jonas Wigander

17 Scaling Agile Mechatronics: An Industrial Case Study 211
Jonn Lantz and Ulf Eliasson

Index 223

Part I
Introduction

This part introduces the concepts underlying the entire book. It consists of three chapters. The first chapter introduces the concept of continuous software engineering as well as the Software Center where the research reported on in this book took place. The second chapter introduces the "Stairway to Heaven," the basic framework underlying the book. The framework defines the typical evolution path that companies evolve through in response to competitive pressures and the need to become more agile and responsive to customer needs. The third chapter captures our learnings, accumulated during the first phase of the software center as well as before, in the effective collaboration between industry and academia. We consider accomplishing tangible industrial impact as equally important as academic publications (or, more accurately, as necessary for relevant academic publications). Consequently, it is critically important for us to establish the most effective collaboration mechanisms between industry and academia.

Chapter 1
Continuous Software Engineering: An Introduction

Jan Bosch

Abstract Software-intensive industries are experiencing an unprecedented evolution of, among others, business models, architectures, ways of working, tooling, and deployment. This transformation allows companies to respond much quicker to changes in the market and to build solutions that much more accurately align with customer needs. The Nordic software-intensive systems industry recognized this challenge and partnered with academia to form the Software Center. The role of the Software Center is to significantly accelerate the rate of adoption of these new approaches at the partner organizations. In this chapter, we discuss the industry trends, introduce the Software Center, and provide an overview of the remainder of the book. This book presents the core results of the first phase (2011–2013) of the Software Center.

1.1 Introduction

The software industry is in a state of transition. Up to some years ago, the development life cycle of software was slow and measured in yearly release cycles for on-premise software. In the embedded systems industry, the development cycle of software was dictated by the development cycle for the hardware and the mechanics of the systems. Software was treated as one of the technology components that went into a product, and the system-level life cycle treated all components equally.

During the last decade, this has started to change considerably. In the Web 2.0 and Software-as-a-Service (SaaS) world, the frequency of software release had been increasing since the early 2000s, and 10 years later, several companies were releasing new software multiple times per day. Initially these were companies that could afford to have failed updates as the consequences to users would be limited,

J. Bosch (✉)
Department of Computer Science and Engineering, Chalmers University of Technology, Gothenburg, Sweden
e-mail: Jan@JanBosch.com

J. Bosch (ed.), *Continuous Software Engineering*,
DOI 10.1007/978-3-319-11283-1_1

but over time systems with more and more critical functionality also started to release with the same multiple times per day frequency.

The Web 2.0 and SaaS companies realized that if something is difficult and hard, such as releasing software to customers without quality issues, one should do it often. This forces the organization to fix all the points where "it hurts." The R&D organization might be able to accept a yearly verification cycle where staff spends many hours during a limited period to fix errors and get the quality up to release levels. However, this becomes infeasible when the release frequency goes up and consequently these tasks become automated and iteratively improved.

Once the release frequency of software went up, a second phenomenon surfaced: Web 2.0 companies were starting to adopt A/B testing as technique to determine the optimal implementation of features and functionality in products. A/B testing is concerned with presenting one version of the software to some users and another version to other users. The behavior of both groups of users is compared in order to determine which version (A or B) leads to more desirable outcomes for the company providing the product. Although Web 2.0 and SaaS solutions provide significant advantages over on-premise software, such as removal of computing infrastructure cost and system maintenance, a major advantage often not recognized is that the experimental approach of A/B testing allowed online companies to optimally align their software with customer needs in unprecedented ways. As made popular by the 2002 Standish Group Chaos report [1], in most software systems half of all the features are never used. This inefficiency provides a major explanation of the advantages of SaaS over on-premise software, as SaaS solutions are able to collect data on what customers actually use orders of magnitude more easy than traditional software.

1.2 Software-Intensive Systems Industry

Over the last decade, a quiet revolution has taken place in the world of embedded systems. Whereas for the longest time, software was either nonexistent or a very small part of embedded systems, now company after company is adjusting its R&D investment towards software. Today, companies in the telecom space, such as Ericsson, spend more than 80% of their R&D budget in software. Even in the automotive space, upwards of 70% of all innovation in products is software related.

A second trend that affects these industries is the transition from products to services or at least products and services. Whereas the vast majority of companies in the embedded systems space traditionally sold products, now more and more companies are transitioning to offer their products as a service. In the telecom space, operators frequently outsource the management of their telecom networks to companies such as Ericsson. In the automotive space, the transition from car ownership to having access to a car as a service is gradually taking shape, especially in large cities where parking problems and congested roads severely diminish the mobility value of owning a car. Famously, the Rolls-Royce company no longer sells

jet engines but instead offers its engines as a service where airlines pay by the flight hour.

When a company that traditionally sold products transitions to offering these products as a service, the value patterns for that company change quite fundamentally. In the case of a products business, maximizing product sales is the key revenue driver. This causes a variety of different strategies, including planned obsolescence, driving customers to buy more instances of a product and to discourage upgrading of already sold products. When the company offers its products as a service, the revenue and profit drivers change fundamentally. The fewer products can be deployed to satisfy the terms of the service contract, and the longer the lifetime of these products, the higher the profitability of the company. One important key factor in extending the life of a software-intensive product is to deploy new versions of software that increase the capabilities of products already deployed in the field.

A development that has seen enormous hype over the last 15 years, but that is finally arriving for real, is the Internet of Things concept. All embedded systems that are somewhat valuable are now connected to the Internet. In the telecom space, products such as base stations and mobile phones have been connected due to the very nature of these products. In the automotive space, trucks have been Internet connected for more than a decade, but the connection was mostly used for fleet management. However, now also cars are becoming connected and much more aware of their surroundings. Also in the consumer electronics space, everything from televisions to cameras has network capabilities. Over the coming years, the price point at which it makes economic sense to connect a product will continue to decrease.

Connected software-intensive products obviously allow for the deployment of new software after the product leaves the factory. However, it also allows for a new trend to develop: embedded systems are now able to collect data about their functioning and the way in which they are used by customers that allows for unprecedented insight into product performance, use, and the context in which products are deployed. Often referred to as "Big Data," the software-intensive systems industry is at the forefront of a transition as large as the transition from on-premise software to SaaS in the pure software world.

In addition to the increased awareness and intelligence of connected products, embedded systems companies will now be able to use the same techniques as Web 2.0 and SaaS companies, including A/B testing. Over time, our world will see cars that get safer, more energy efficient, and more user-friendly as the R&D department will continuously deploy new implementations of functionality in groups of cars to test which leads to better outcomes in the real world, rather than in the lab. This also means that the product R&D process will see a fundamental split between anything "atoms" and anything "bits." Products start to hit the market with only a basic set of software features that will grow and improve over time.

Of course, many obstacles to realizing this future exist. These include security and safety of potentially life-threatening products, certification of safety-critical systems that will slow down the frequency at which software can be updated,

bandwidth constraints, as well as many others. However, in time these obstacles will be overcome and the future as we outline here will materialize.

The software-intensive systems industry stands at the beginning of fundamental shift in business models, architectures, technologies, ways of working, and customer interaction. As with every industry transformation, there will be winners and losers among the incumbents and major opportunities for new entrants. It's going to be interesting.

1.3 The Changing Practice of Building Software

As the Web 2.0 and SaaS world started to exploit the new capabilities of continuously deploying software on their servers and were able to collect data on how customers were using the data, the practices around software development started to change as well. These changes coincide to a large extent with the agile development practices around teams, verification and validation, and deployment.

Traditional software development is organized sequentially, handing over intermediate artifacts such as requirements, architecture designs, software, testing results, etc. between different functional groups in the organization. This caused a significant number of handover points to exist in the organization that caused three main problems: First, there are significant time delays associated with the handovers between different groups in the organization that provide the intermediate artifacts. Second, significant amounts of resources are applied to creating these intermediate artifacts that, to a large extent, are replacements of human-to-human communication. Third, the functional organization tends to cause significant local optimization in the respective teams and, especially when clear data and feedback from the customer are missing, may result in a politicized organization.

In agile software development, the notion of cross-functional, multidisciplinary teams plays a central role. These teams have the different roles necessary to take a customer need all the way to a delivered solution or solution extension delivered to a customer. Depending on the maturity in the organization (in the next section we present a model), the team may have product management, software architecture, development, user experience, testing, delivery, as well as customer service skills present, although individuals may combine multiple roles and bring multiple skills to the team.

In the area of software release, a similar transformation has taken place. In the waterfall development process, the final stages before the release of software are dictated by weeks and sometimes months of verification and validation activities. Significant numbers of staff are taken out of their normal jobs and dedicated to the effort of getting the product out the door and in the hands of customers. Especially in the area of software-intensive embedded systems, the cost of errors surfacing in the field after the product has left the factory can be very high, and consequently, companies spend significant amounts of time and effort to minimize the risk. Obviously, considering the cost and effort, companies are inclined to release

software as infrequently as possible, resulting in slow release cycles with major chunks of new functionality being deployed at every release.

One of the memes in agile software development is "if it hurts, do it often." Consequently, the incredibly painful process of releasing new software falls under that motto. This led to the development of continuous integration and continuous delivery practices. There are two main changes in this approach. First, the developers themselves, rather than testers, are responsible for creating the test cases for their software and providing these test cases to the continuous integration environment. This meshes very nicely with test-driven development practice in most agile development practices. Second, increasing the frequency of integration and release requires a major increase in the level of automation. Although many organizations insist on a final human-in-the-loop check before deploying new software, much of the process can be automated with proper prioritization by R&D management.

1.4 A Systematic Evolution Model

As we outlined in the previous sections, there are fundamental changes on the horizon for the software-intensive systems industry. Also, as we discussed in the previous section, the Web 2.0 and SaaS industries have incorporated major changes in the way software is developed. Many organizations that we have worked with, however, struggle with the correct order in which to implement the changes in the organization. Especially for management that has very strong domain knowledge, but is perhaps no expert in the intricate details of software engineering, it can be difficult to determine how to transition from the current state to the bright new future approach that is presenting itself to temptingly on the horizon. In many ways, it is easy to know where to go, but it is hard to determine the optimal path to getting there.

In response to this challenge, we have developed a model that describes the typical successful evolution of a company from traditional development to customer data-driven development. We named the model the "Stairway to Heaven." A lofty name but one that has become a meme in the companies that have adopted the model for their change journey. In Fig. 1.1, the model is shown graphically.

The Stairway to Heaven model is described in detail in the next chapter, so in this chapter we provide a high-level introduction into the model. The model

Fig. 1.1 The "Stairway to Heaven" evolution model

consists of five steps. Starting with traditional development, the first step is the adoption of agile work practices in the R&D organization. These practices include, among others, the notion of small, empowered teams, the backlog, and daily stand-ups and sprints.

Once agile work practices have been adopted, typically the teams will start to express their frustration with their inability to test the code that they build in the broader context of the system. This leads to the adoption of continuous integration as the next set of practices in the organization. These practices include test-driven development by agile teams, an automated build and test environment, and clear definition of functional, legacy, and quality requirements of the system or system family.

During our work with the software-intensive systems industry, we have, at several occasions, experienced a situation where a B2B client of a company realized the maturity of the continuous integration approach employed by the company. Upon realizing this, the client then requests or, depending on the power relationship, demands more frequent releases of software for the hardware that it acquired from the company. This is when the company starts to prepare to enter the next stage: continuous deployment where software, at the end of agile sprints or even more frequently and after passing the continuous integration testing activities, is deployed at customers. One important transition at this point is that whereas earlier the client decided when to install a new version of the software, when adopting continuous deployment, it is the company that decides when its customers move to the newer version of the software. This requires a level of quality control in most companies that was not in place in earlier stages of the Stairway to Heaven model.

The final stage is where the company realizes that the frequent deployment of software to its customers can be used for the continuous testing of new features as well as the optimization of existing features. This is where software-intensive systems companies start to realize that the benefits so far exclusive to Web 2.0 and SaaS companies have also become available to them. As we mentioned earlier in this chapter, half of the features companies put in products are not used. Finally, also software-intensive systems companies have mechanisms available to them to determine which of the features are used and which are waste and might as well be removed.

We have described the five stages of the Stairway to Heaven model in sequence and, based on our experience of the dozens of companies that we have worked with, we have strong evidence that organizations evolve their software engineering practices in this order. Even reasoning logically allows one to determine that this order is the only suitable order: adopting A/B testing and other approaches to test functionality with customers cannot be accomplished without continuous deployment of software. No company will adopt continuous deployment without having its continuous integration practices, tooling, and automation in place. Continuous integration has no business value if the organization does not adopt agile work practices. Traditional waterfall development assumes late, "big-bang" integration of software by different component teams, invalidating the relevance of continuous

integration. Consequently, adopting agile work practices is the starting point for any adoption of new software engineering practices.

1.5 Software Center

During the summer of 2011, after more than a year of preparation, the Software Center kicked off its research activities. The Software Center started as a partnership between two universities and four companies but has over the first years of its existence grown to ten partners. At the time of writing, these partners include seven companies and three universities. The companies are Ericsson, Volvo, Volvo Car Corporation, Saab Electronic Defense Systems, Axis Communications, Grundfos, and Jeppesen, a daughter of Boeing. The universities are Chalmers University of Technology, University of Gothenburg, and Malmö University. Chalmers is formally the host of the Software Center.

The center was formed on the request by the founding companies: Ericsson, the Volvo companies, and Saab. These companies realized that software was becoming an increasingly important and differentiating part of their respective products. This requires software engineering to become a core capability for these companies. By definition, a core capability requires world-class proficiency, and although the companies were successful in developing the software part of their products, engineering software needed to become a stronger skill at these companies. A second realization of these companies was that the amount of software in their products was growing by an order of magnitude every 5–10 years, depending on the industry, and it was not feasible to hire that many software engineers, neither from a practical hiring perspective nor from an economic perspective. Consequently, the ability to develop high-quality software that was growing continuously with current or slowly growing staff numbers and much faster than the companies had required up to that point was recognized as critical.

The goal of the Software Center, consequently, can be summarized as "10X in 10 years," i.e., ten times the productivity in 10 years. The Software Center helps companies to accelerate the adoption of best practices in software engineering, a field where best practices are evolving rapidly.

The center consequently has a dual goal, i.e., academic excellence and tangible impact at the partner companies. Academic excellence was demanded by the companies, as there is little point in working with researchers that are not at the forefront of their field. This would cause the companies to adopt practices that were not in their best interest and might cause them to get behind their competitors. At the same time, tangible impact at the partner companies was demanded as knowing the best software engineering practices, but not employing these practices at the companies would do nothing to improve the competitive position of the companies.

Over the last years, the researchers in the Software Center have partnered closely with the practitioners at the partner companies to study a number of topics, and the results are presented in this book. The overall conceptual model for the center has

been the Stairway to Heaven, described in the previous section. As the companies in the center are in different points in the model, this has meant that the companies had different challenges to address. However, there was sufficient overlap in the challenges of the companies to allow the researchers to study each challenge at multiple companies.

Research in the center was performed using sprints. Although sprints in agile software development are 4 weeks or less, in the research activities we have adopted a 6 month heartbeat. This means that each research project operates on a 6 month life cycle where during the 6 months, the project goes through all the stages of formulating the next part of the research problem to study, designing the study, collecting data at the companies, analyzing the data, reporting results to the industrial partners, and reporting to the academic community. Although adopting a sprint model raised concerns with some of the researchers, interestingly enough we found similar benefits of using agile practices in research as one finds in industry: as researchers study a smaller slice of a large, complex problem, we learned more about the large problem iteratively, which allowed for more frequent steering of research by both the companies and the researchers alike.

The advantages of the model that we have adopted in the Software Center are multiple. For companies, the approach provides a more consistent, integrated focus on their key change initiatives. In addition, the research takes a holistic approach including technical, organizational, and business aspects, rather than narrowly focusing on a technical area. Also, companies receive value from their participation in the center every 6 months and have the opportunity to steer projects frequently.

For researchers, the primary advantage is that the close partnership with industry allows one to implement and study new approaches to software engineering empirically. As software engineering research is becoming increasingly empirical in nature, the danger is that only the status quo at companies can be studied, as no data exists for novel approaches. Research in the context of the center allows researchers to introduce and experiment with novel approaches in the context of software development organizations and to collect empirical data on the benefits that these approaches might offer.

1.6 Structure of This Book

This book reports on the results of the Software Center research and collaboration during its first phase, 2011–2013. It captures the evolution and changes in the Software Center companies, and as such it provides an insightful perspective on the adoption of modern software engineering practices in large, software-intensive systems companies where hundreds or even thousands of engineers collaborate to deliver on new systems and new versions of already deployed systems.

Although Web 2.0 and SaaS companies often are ahead of other industries, it is important to realize that many of these companies are much smaller in terms of staff and hence experience fewer scaling problems. In addition, the software at these

companies is deployed at the company's own servers instead of the physical product that is in the hands of customers. This simplifies the practical as well as organizational, legal, and customer relation constraints experienced by companies that put a physical product in the hands of customers.

Many software-intensive system companies are on the same change journey as the Software Center companies or about to embark on the change journey outlined in the section on the Stairway to Heaven. This book is intended for the practitioners at those companies and provides concrete models, frameworks, and case studies that show the specific challenges that the partner companies experienced as well as the solution approaches employed and the outcomes of using those approaches.

The book is structured primarily based on the Stairway to Heaven model. The first part, in which this chapter is part of, is concerned with the introductory parts. The next chapter describes the Stairway to Heaven in much more detail. The chapter after that, by Anna Sandberg, describes our learnings from collaboration between industry and academia. These learnings are based on the experiences in the Software Center as well as the engagements between Ericsson and Chalmers University of Technology in the years before. In total more than a decade of industry-academia collaboration is captured in that paper.

Part II is concerned with step 2 in the Stairway to Heaven where R&D is in the process of adopting agile work practices. One of the key challenges in large-scale agile is that although teams are, ideally, as independent and empowered as possible, in practice significant amount of interaction takes place between teams. Our research shows that especially these inter-team interactions slow development down, and one of the chapters studies this and provides a set of tactics to minimize inter-team delays. A second major area in large-scale agile development is to maintain the architecture as cross-functional feature teams are mostly concerned with developing their features and less with the long-term viability of the architecture. A second chapter studies the current role of architects in agile teams and improvements that can be implemented to empower architects in agile development. Large-scale development, agile or traditional, typically uses platforms and other forms of software reuse to focus the development effort on the most differentiating functionality for customers. The challenge is that teams working with reusable software tend to be less focused on speed, causing the teams reusing the software platform to be slowed down as the software platform organization is unable to keep up with the requests from agile teams. This easily results in a situation where the overall development speed of the organization, measured from the identification of a customer need to the delivery of a solution meeting that need to customers, is not improved. The third chapter in this part studies enablers and inhibitors for combining development speed with high levels of reuse. The final chapter in part II studies a team-based approach to increasing responsiveness to individual customers while maintaining high throughput of road map features to the entire customer base. The paper describes the notion of customer-specific teams where some of the dozens of agile teams are dedicated to specific customers. These customer-specific teams only implement features requested by "their" customer, but the software built by the team is integrated

into the main software baseline and the same quality procedures are applied to this code as to road map features.

Part III of the book combines two phases of the Stairway to Heaven, i.e., continuous integration (CI) and delivery (CD) as the two phases are highly interconnected. The part is concerned with four aspects concerning CI and delivery. First, techniques to develop an understanding of and for visualizing end-to-end testing activities across the organization. In our research, it became clear that few of the staff at the Software Center companies had a full, end-to-end understanding of all the testing activities that took place in the organization. The first paper presents a visualization technique that clearly shows all testing activities, the scope of testing, as well as the frequency of testing. Second, our research showed that the build system infrastructure and the integration flows differed significantly between different partner companies but that there was limited insight into the consequences of different approaches. The second paper aims to categorize the dimensions of variation and the consequences of different choices. As it is clear from our research that testing takes place in many stages and scopes during software development, we need to study how to perform partial testing while reaching conclusions concerning quality that are reliable. The third paper is concerned with partial testing and its consequences. Finally, the last paper in part III is concerned with automatic testing of graphical user interfaces. This is a notoriously difficult and effort-consuming area of testing as GUIs tend to change frequently, and traditionally, test cases were extremely brittle. This causes a level of test case maintenance cost that did not warrant for automated testing. However, as the field progresses, novel approaches have evolved that address these limitations, and the final paper in this part describes these approaches.

Part IV is concerned with the highest level of the Stairway to Heaven, referred to R&D as an innovation system. As described earlier in this chapter, once continuous deployment and data collection are in place, companies can start to experiment with new features rather than build everything in slow, product management-driven road maps. The part holds two chapters. The first describes the current state of data collection in the Software Center companies as well as the use of this data in R&D. The second paper presents the hypothesis/experiment approach to software development, i.e., the HYPEX model. This paper codifies many of the concepts that we have discussed in this chapter in a systematic approach and discusses some cases from companies applying some or all of the practices of the HYPEX model.

Part V of the book addresses a topic that is orthogonal to the Stairway to Heaven and yet critically important in large organizations: organizational performance metrics that capture data and visualizations of the status of software assets, defects, and teams. The first chapter describes approaches to profiling software products as well as profiling R&D organizations. The data collected for profiling as well as the profiles themselves can be used to direct R&D effort, testing activities, perform risk assessments, as well as a variety of other purposes. The second chapter is concerned with implementing measurement systems in software-intensive systems companies that can stand the eroding effect of time. Over time, changes in the organization occur, causing traditional measurement systems to break down or to present

unreliable data. The chapter introduces the notion of self-healing systems that adjust themselves to architectural, infrastructural, and organizational changes.

The final part of the book captures the perspective of two of the partner companies. The first chapter captures the experiences of the change journey at Ericsson. The team writing the chapter has worked with dozens of R&D teams across the company and has distilled the learnings from those teams. The second chapter sheds light on the experiences at Volvo Cars, especially concerning the adoption of model-driven engineering in the context of climbing the Stairway to Heaven as well as the changing relationship between the company and its supplier base.

Conclusions
Software engineering for software-intensive embedded systems is in a major state of change. Trends such as the growing importance of software, the customer expectation of responsiveness, the connectivity becoming available to virtually all embedded systems, as well as modern, agile software engineering practices are about to fundamentally disrupt how we build large systems. A transformation similar to the transition from on-premise software to SaaS and Web 2.0 is about to disrupt the embedded systems industry. Software Center, a partnership between seven global companies and three universities, aims to proactively support this transformation and to accelerate the change that the partner companies go through with the intent of maintaining and extending the competitive position of these companies. This book captures the key learnings from the first phase of the Software Center as the companies climb the Stairway to Heaven. The book captures the models, frameworks, and tools developed by the researchers and practitioners in the center as well as the experiences from the partner companies in applying these novel software engineering techniques.

Reference

1. http://theagileexecutive.com/2010/01/11/standish-group-chaos-reports-revisited/

Chapter 2
Climbing the "Stairway to Heaven": Evolving From Agile Development to Continuous Deployment of Software

Helena Holmström Olsson and Jan Bosch

Abstract Software-intensive systems companies need to evolve their practices continuously in response to competitive pressures. Based on a conceptual model presented as the "Stairway to Heaven," we present the transition process when moving from traditional development towards continuous deployment of software. Our research confirms that the transition towards agile development requires a careful introduction of agile practices into the organization, a shift to small development teams, and a focus on features rather than components. The transition towards continuous integration requires an automated test suite, a main branch to which code is continually delivered, and a modularized architecture. The move towards continuous deployment requires internal and external stakeholders to be fully involved and a proactive customer with whom to explore the concept. Finally, the transition towards R&D as an innovation system requires careful ecosystem management in order to align internal business strategies with the dynamics of a competitive business ecosystem. Characteristic for all transitions is the critical alignment of internal and external processes in order to fully maximize the benefits as provided by the business ecosystem of which a company is part.

2.1 Introduction

Today, software development is conducted in increasingly dynamic environments characterized by inter-organizational relationships and dependencies. From being an activity defined by an organization's internal processes and practices, software

H.H. Olsson (✉)
Department of Computer Science, Malmö University, Malmö, Sweden
e-mail: helena.holmstrom.olsson@mah.se

J. Bosch
Department of Computer Science and Engineering, Chalmers University of Technology, Gothenburg, Sweden
e-mail: Jan@JanBosch.com

J. Bosch (ed.), *Continuous Software Engineering*,
DOI 10.1007/978-3-319-11283-1_2

development is transitioning towards becoming an activity characterized by open innovation and co-creation of value in which an ecosystem of stakeholders co-evolves capabilities around new innovations [1, 2]. Typically, fast-changing and unpredictable market needs, complex and changing customer requirements, and pressures of shorter time-to-market are challenges that face most business ecosystems. To address this situation, the vast majority of companies have started adopting agile development methods with the intention to enhance the organization's ability to respond to change. In emphasizing flexibility, efficiency, and speed, agile practices have led to a paradigm shift in how software is developed [3–5].

However, while the adoption of agile practices has enabled companies to shorten development cycles and increase customer collaboration, there is an urgent need to learn from customers also after deployment of the software product. The concept of continuous deployment, i.e., the ability to deliver software more frequently to customers and benefit from frequent customer feedback, has become attractive to companies. If this is achieved, companies could benefit from even shorter feedback loops, more frequent customer feedback, and the ability to more accurately validate whether the functionality that is developed corresponds to customer needs and behaviors. While the ability to continuously deploy new software functionality creates new business opportunities and extends the concept of agile practices, it presents a number of challenges.

In this chapter, we present a multiple-case study in which we explore four software development companies that are currently moving towards continuous deployment of software. We present the challenges these companies face when transitioning towards continuous deployment and beyond. Also, we identify actions companies need to take to address, and overcome, these challenges. As our theoretical framework for analysis, we use the ESAO model [6]. The model provides dimensions for assessing the end-to-end dimensions of business, technology, and organization with considerations especially taken to external stakeholders and the business ecosystem of which a company is part.

2.2 Background

2.2.1 The "Stairway to Heaven"

Companies evolve their software development practices over time. Typically, there is a pattern that most companies follow as their evolution path and that we have reported on in previous studies [5]. We refer to this evolution as the "Stairway to Heaven," and it is presented in Fig. 2.1. The phases of the "Stairway to Heaven" are discussed in more detail in the remainder of this section. As a summary, companies evolving from traditional development start by experimenting with one or a few agile teams. Once these teams are successful, agile practices are adopted by the R&D organization. As the R&D organization starts showing the benefits of working

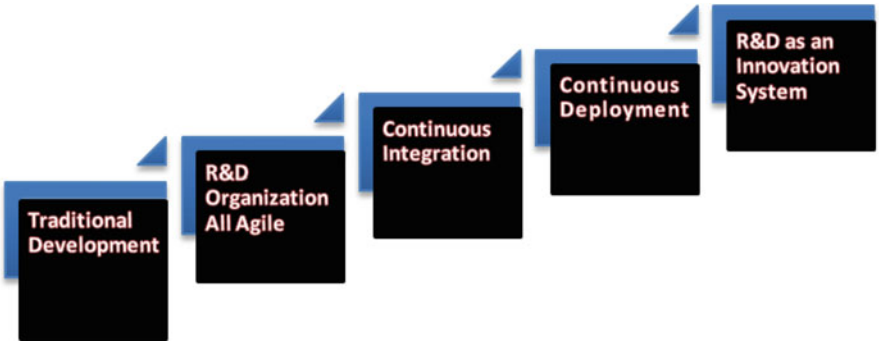


Fig. 2.1 The "Stairway to Heaven" evolution model

agile, system integration and verification becomes involved and the company adopts continuous integration. Once continuous integration runs internally, lead customers often express an interest to receive software functionality earlier than through the normal release cycle. They want continuous deployment of software. The final step is where the software development company collects data from its customers and uses the installed customer base to run frequent feature experiments.

Step 1: As the first step in our model, and the starting point for most embedded systems development companies, there is traditional development. We refer to traditional development as an approach to software development characterized by slow development cycles, sequential phases, and a rigorous planning phase in which requirements are frozen upfront [7]. Typically, the approach is characterized by a waterfall-style interaction between product management, product development, system test, and the customer. Projects adopting this development approach suffer from long feedback cycles and difficulties to integrate customer feedback into the product development process [5, 7]. Typically, delivery to the customer takes place in the end of the project life cycle, and it is not until then that customers can provide feedback on the software functionality they have received.

Step 2: As a way to address the many challenges experienced in the traditional development approaches, most companies start adopting agile development practices. Agile practices are characterized by small cross-functional development teams, short development sprints resulting in working software, and continuous planning in which the customer is involved to allow for continuous customer feedback [4, 8]. In agile organizations, however, product management and system verification still work according to the traditional development approach [5].

Step 3: The third step in our model is where a company succeeds in establishing practices that allow for frequent integration of work, daily builds, and fast commit of changes, e.g., automated builds and automated test. At this point, both product development organization and test and verification organization work according to agile practices with short feedback cycles and continuous

integration of work. Humble and Farley [9] define continuous integration as a software development practice where members of a team integrate their work frequently, leading to multiple integrations per day.

Step 4: Continuous deployment implies that you continuously push out changes to the code instead of doing large builds and having planned releases of large chunks of functionality. This allows for continuous customer feedback, the ability to learn from customer usage data, and to eliminate work that doesn't produce value for the customer. At this point, R&D, product management, and customers are all involved in a rapid, agile development cycle in which response time is short [10].

Step 5: The fifth and final step in the "Stairway to Heaven" evolution model is where the development organization responds based on instant customer feedback and where actual deployment of software functionality is seen as a way of validating functionality. The intention is to expose customers to partial implementation of functionality and use their feedback for determining the value of that particular functionality [11].

2.2.2 The ESAO Model

One of the most common models to provide a holistic perspective of the dimensions of business, technology, and organization is the BAPO model [12, 13]. The BAPO model defines four concerns, i.e., business, architecture, process, and organization, which can be used for the mapping of roles and responsibilities and for understanding organizational structures [13]. The model is frequently applied for analysis and assessment in both academia and industry.

More recently, the ESAO model has been proposed as an extension to the BAPO model [6]. The ESAO (Ecosystem, Strategy, Architecture and Organizing) model consists of six interdependent dimensions that are important to take into account for software development companies when assessing the alignment between their business, their technologies, their processes, and their internal and external interactions. The six dimensions concern an internal and an external company perspective, and it is helpful for understanding the interdependency between a company and the ecosystem of which it is part. In this study, we apply the external dimensions, i.e., the ecosystem dimensions, in our analysis of the challenges and the actions involved in the transition between each of the steps in the "Stairway to Heaven" model.

In Fig. 2.2, we present the ESAO model. The model emphasizes that the internal and external perspective on each dimension need to be aligned with each other, i.e., ecosystem strategy and internal strategy, ecosystem architecture and internal architecture, and ecosystem organizing and internal organizing. The ESAO model focuses on achieving and maintaining alignment between internal and external dimensions, and therefore, we find it useful in our analysis of how actions required by a company interplay with the ecosystem of which the company is part.

Fig. 2.2 The ESAO model [6]

2.2.2.1 The Ecosystem Perspective

In the ESAO model, the ecosystem perspective emphasizes the role of an organization in a larger context and the implications this has on business strategy. As illustrated in the model, it is critical to align internal processes with external processes and to align internal change initiatives with strategic options that are within the business ecosystem of which the organization is part. Below, we describe each dimension as presented in [6]:

Ecosystem Strategy: The strategy dimension of a company is related to the business ecosystem of the organization and the strategic options that it has available in its role in this ecosystem. Depending on the strategic choices made by the company, there are significant implications on the system and software development practices of the organization.

Ecosystem Architecture: The architecture dimension defines the interface between an organization's internal architecture and the solutions that are provided by suppliers, firms that build software on top of a product or platform, and other roles that have an impact on the product and its architecture. In addition to the focus on interfaces between different stakeholders, focus is also on the architecture strategy. In being part of a business ecosystem, commoditization and innovation of new functionality are ongoing processes that have an impact on long-term architectural planning.

Ecosystem Organizing: The organizing dimension deals with how firms work with their customers, suppliers, and ecosystem partners in terms of processes, tools, and ways of organizing the communication and collaboration. For example, many companies have internally adopted agile ways for working while they have suppliers that still use traditional development, causing a disruption in the internal

development processes. In the ESAO model, the organizing dimension emphasizes the importance of aligning internal and external development and release processes.

2.3 Research Method and Sites

This study builds on a six months multiple-case study [14], involving four software development companies. All four companies are moving towards continuous deployment of software. In representing different stages in this transition, we find these companies of particular interest for understanding the challenges involved, as well as the actions to take, in this process.

The main data collection method used is semi-structured interviews with open-ended questions [15]. In total, 18 interviews were conducted. In companies A and B, we conducted five interviews in each company, involving software and function developers, software architects, system engineers, configuration managers, and project leaders. In companies C and D, we conducted four interviews in each company, involving software developers, component/system integrators, project/release managers, product line maintenance, and a product owner. All 18 interviews were conducted in English and each interview lasted for about 1 h. In addition to summarizing notes, all interviews were recorded in order for the research team to have a full description of what was said [14]. Each interview was transcribed and the transcriptions were shared among the researchers to allow for further elaboration on the empirical material. In addition to the interviews, documentation review and field notes were complementary data collection methods, including software development documents, project management documents, and corporate websites and brochures. The four companies involved in our study are described below:

Company A is involved in developing systems for military defense and civil security. The systems focus on surveillance, threat detection, force protection, and avionics systems. Internally, the company is organized in different departments with systems engineering (SE) and quality assurance (QA) being the two departments included in this study. In relation to the model presented in Fig. 2.1, this company is best described as a company doing traditional development but moving towards an agile R&D organization.

Company B is an equipment manufacturer developing, manufacturing, and selling a variety of products within the embedded systems domain. The company structure is highly distributed with globally distributed development teams. Also, suppliers do a significant part of the development. In relation to the model presented in Fig. 2.1, this company is described as a company close to continuous integration. Still, parts of the organization are traditional, but there are a number of teams that operate in a highly agile manner and that have continuous integration in place.

Company C is a manufacturer and supplier of transport solutions for commercial use. Similar to company B, the development organization is largely dependent on supplier organizations. In relation to the model presented in Fig. 2.1, this company can be described as a company with parts of its development organization

being traditional and parts of it being highly agile and with continuous integration practices in place.

Company D is a provider of telecommunication systems and equipment, communications networks, and multimedia solutions for mobile and fixed network operators. The organization is highly distributed with globally distributed development. In relation to the model presented in Fig. 2.1, this company is described as a company with established practices for continuous integration and with initiatives to continuous deployment in place.

2.4 Findings and Analysis

In this section we present our case study findings and analysis. In our presentation, we outline the challenges our interviewees experience when transitioning between each of the steps in the "Stairway to Heaven" (Fig. 2.1). In our analysis, we use the ESAO framework (Fig. 2.2) and the dimensions of (1) ecosystem strategy, (2) ecosystem architecture, and (3) ecosystem organizing, to describe each transition and what actions the companies need to take to address these challenges.

2.4.1 From Traditional to Agile R&D

All companies involved in our study are transitioning towards agile development practices. Either small parts of the organization are agile or, as in some companies, the majority of the development teams work according to agile practices. One of the challenges that the companies experience when transitioning towards agile practices is that current processes and ways of working are often difficult to align with the agile parts of the organization. Also, while internal processes and structures are agile, it is not necessarily so that external stakeholders, such as suppliers and customers, transition towards agile practices. To be part of a larger business ecosystem has implications on processes as well as tools, and this is something that all companies experience as a true challenge when going agile. Another challenge is the business model that is usually traditional in character and based on yearly releases and long development cycles. This gives a conservative impression with expectations set upfront rather than being flexible and responsive towards evolving customer needs.

Based on the experiences from our study, we outline actions companies need to take when transitioning from traditional to agile development. To do this, we use the ESAO ecosystem dimensions [6] emphasizing the importance of aligning internal company dimensions with external ecosystem dimensions.

Ecosystem Strategy: The business strategy is critical when establishing a culture and support for a new development approach. Our study emphasizes the importance of introducing agile development as an approach that allows for new business

opportunities in terms of frequent releases, shorter time-to-market, and close collaboration to customers. While this creates new opportunities in terms of business value, it might upset other stakeholders in the ecosystem. Therefore, companies need to strategically position themselves to maximize revenue while at the same time contribute to the business ecosystem.

Ecosystem Architecture: When transitioning towards agile development, an important initiative is to make sure that feature teams are supported by an architect who "safeguards" the team and the integrity of the architecture. Our study emphasizes the importance of having appointed architects that work closely with the development teams and help in protecting the architecture. Also, these architects play an important role in aligning the internal architecture with the dynamics of an external ecosystem.

Ecosystem Organizing: The introduction of agile working practices is important and requires strong managerial support. Agile development implies small, cross-functional teams working on parts of the functionality. One important initiative when moving towards agile development is therefore to have mechanisms for team formation and for teams to empower and self-direct themselves. Also, renegotiating supplier contracts to facilitate for agile development might be necessary and, if so, needs to be a highly prioritized activity.

2.4.2 From Agile to Continuous Integration

Several of the companies involved in this study have continuous integration practices in place in at least parts of the organization. While the adoption of automated tests is diverse, it is seen as the way forward and as one highly prioritized activity to achieve more frequent delivery of software. According to our interviewees, there are a number of challenges to address when transitioning from agile to continuous integration. First, the dependency to other ecosystem stakeholders such as suppliers makes development complex, and most interviewees experience this dependency as having a negative effect on development speed. In addition, several of the interviewees mention that fitting different components from different suppliers takes time, so it is not only the development lead-time that is long but also the integration of components that is time-consuming and costly. One challenge that is often mentioned is the dependency between components and component interfaces. This makes modularization difficult and hence, development teams are highly dependent on each other. Another challenge is the testing activities. All companies emphasize the need for automatic testing and daily builds, while at the same time finding this is difficult in an embedded system involving hardware with slow development cycles.

Based on the experiences from our study, we outline actions companies need to take when transitioning from agile to continuous integration. To do this, we use the ESAO ecosystem dimensions [6] emphasizing the importance of aligning internal company dimensions with external ecosystem dimensions.

Ecosystem Strategy: When transitioning to continuous integration, the business perspective needs to shift from the "milestone perspective" with yearly releases to a perspective in which delivery and release are viewed as continuous activities and where there is always a shippable product. This requires support, not only from the internal business models but also from other stakeholders in the business ecosystem, such as external suppliers as well as customers.

Ecosystem Architecture: To reap the benefits of continuous integration, development needs to be modularized into smaller units, i.e., the build process needs to be shortened so that tests can be run more frequently on particular parts of the system. Our study shows that development lead-time is efficiently reduced once the architectural concerns are modularized. This allows for more frequent deliveries, the opportunity to learn more frequently from feedback, but also for efficient decoupling of dependencies to other ecosystem stakeholders.

Ecosystem Organizing: The transition towards continuous integration requires a cultural shift and a transformation of previous traditions and values. Companies need to organize themselves to allow for short cycles between the development organization and validation and verification. To align these two is critical to benefit from daily builds and frequent tests. Also, companies need to adopt test-driven development practices and processes that support automated tests. In order to reduce complexity, companies should strive for a process in which code is checked into one main development branch, i.e., the production line, to avoid having several branches since that will only add to complexity and lead-time.

2.4.3 From Continuous Integration to Continuous Deployment

In several of the companies involved in this study, continuous integration is a well-established practice, and there are attempts to involve proactive customers in continuous deployment of software functionality. However, there are a number of barriers that need to be addressed to succeed in this endeavor. What is most evident from our interviews is the complexity that arises in different network configurations at customer sites. While the product has its standard configurations, there are always customized solutions as well as local configurations that add complexity. In a business ecosystem involving a large number of stakeholders, this becomes a major task to manage for the company deploying the software. A common challenge is when a customer wants a new feature but has an old version of the product to which this new feature has to be configured. From the interviews it is clear that customers still regard upgrades and new features as a challenge due to the risk of interfering with legacy. Furthermore, the internal verification loop needs to be significantly shortened to not only develop functionality fast but to also deploy it as fast at customer site.

In the transition towards continuous deployment, all corporate functions need to be involved. Similar with introducing agile practices to different parts of the organization as the very first step when moving from traditional development to agile development, the transition towards continuous deployment requires involvement of different organizational units in order to fully succeed. Especially, product management needs to be involved as they are the interface towards customers. Finally, finding a proactive customer who is willing to explore the concept of continuous deployment is found critical.

Based on the experiences from our study, we outline actions companies need to take when transitioning from continuous integration towards continuous deployment. To do this, we use the ESAO ecosystem dimensions [6] emphasizing the importance of aligning internal company dimensions with external ecosystem dimensions.

Ecosystem Strategy: When moving towards continuous deployment of software, companies need to identify a proactive customer within the ecosystem who is willing to explore the concept. When having identified a "lead customer," the development organization can start building a continuous deployment culture and capability, in which fast customer feedback serves as direct input to improved software functionality. The lead customer serves as a role model to other customers in the ecosystem and benefits from the opportunity to get new software functionality as soon as the development organization has something to deliver. To achieve this, companies need to adapt their business model and strategy to support continuous deployment of functionality. Also, continuous deployment of software will allow companies to move closer to its customers, and therefore, current relationships and dependencies within the ecosystem need to be strategically reconfigured.

Ecosystem Architecture: When deploying software on a continuous basis, companies need efficient mechanisms to roll back unsuccessful deployments and mechanisms to deploy parts of the system rather than the entire system. In terms of architecture, complexity arises when different customers have different network configurations to which software is deployed. While the product has its standard configurations, there are always customized solutions as well as local configurations that cannot be fully accounted for. Therefore, companies need to carefully align the internal architecture with the ecosystem of which it is part.

Ecosystem Organizing: To succeed with continuous deployment, all corporate functions need to work in short cycles and with the intention to deliver smaller features more frequently to customers. This requires that not only the development and testing organizations work in short cycles but that product management and release, as well as customers, engage in this. Similar with introducing agile practices, the transition towards continuous deployment requires involvement of different organizational units in order to fully succeed. In addition, external ecosystem stakeholders are affected which requires companies to manage increased complexity and competitiveness.

2.4.4 From Continuous Deployment to R&D as an "Innovation System"

In one of the companies involved in this study, agile processes have been around for several years, and they have become widespread within the company. A large part of the organization is familiar with continuous integration, and the company is pushing towards continuous deployment for parts of the product and for a selected segment of its customers. The fast feedback loop to customers is regarded as the major benefit. According to the company, faster feedback means cheaper development since the development organization can spend time on developing things customers want instead of correcting mistakes in functionality that is not necessarily what the customer asked for. In advancing towards R&D as an "innovation system," the company foresees that their product needs to be instrumented so that customer data can be automatically collected from the installed product base. To identify relevant metrics is a highly prioritized activity in order to collect data that will work as a basis for product improvements and new feature development. Second, the company needs to develop the capability to effectively use the collected data to test new ideas with customers. In the transition towards R&D as an innovation system, the company will move closer to its customers. This might upset other stakeholders in the ecosystem who have previously been the ones interacting with customers. To carefully manage this tension, while at the same time generate business revenue, is a challenge identified as critical for successful co-creation of customer value.

Based on the experiences from our study, we outline actions companies need to take when transitioning from continuous deployment to R&D as an "innovation system." To do this, we use the ESAO ecosystem dimensions [6] emphasizing the importance of aligning internal company dimensions with external ecosystem dimensions.

Ecosystem Strategy: To have R&D as an innovation system that responds directly based on customer feedback, companies need to establish a strategy and culture in which they continuously innovate in synergy with its customers. Instead of having requirements frozen before development starts, this approach allows for evolving requirements once the system is taken into use by its customers. To achieve this, strategic collection and analysis of customer data are critical, as well as mechanisms that allow for quick response to customers. Also, business and pricing models need to support short-cycle and customer data-driven innovation processes in the ecosystem.

Ecosystem Architecture: To support a viable ecosystem architecture, infrastructures need to be established that support run-time variation of functionality. This is required to allow for continuous innovation and experimentation with customers. For example, the architecture needs to support A/B testing which is one technique that is applied by companies when experimenting with customers. Also, instrumentation of code for data collection purposes is necessary which usually requires an extension of current architectures.

Ecosystem Organizing: For achieving short cycles and rapid response to customer feedback, requirements, development, validation, and release functions need to work together. Initiatives that facilitate for aligning and integrating internal and external corporate functions need to be prioritized. Companies need to develop the capability to effectively use data collected from other ecosystem stakeholders in order to improve current system versions, develop new functionality, and innovate entirely new products. This requires careful collaboration with other ecosystem stakeholders and an organization that allows external contributions.

Conclusions
In this study, we explore how software development companies evolve their practices over time. Based on a conceptual model presented as the "Stairway to Heaven," we present the transition process when moving from traditional development towards continuous deployment of software.

Based on our analysis, and in accordance with previous research, we see that the transition towards agile development requires a careful introduction of agile practices into the organization, a shift to small development teams, and a focus on features rather than components [16]. The transition towards continuous integration requires an automated test suite, a main branch to which code is continually delivered, and a modularized architecture [9]. The move towards continuous deployment requires internal and external stakeholders to be fully involved and a proactive customer with whom to explore the concept [9]. Finally, the transition towards R&D as an innovation system requires careful ecosystem management in order to align internal business strategies with the dynamics of a competitive business ecosystem [6, 11]. Characteristic for all transitions is the critical alignment of internal and external processes in order to fully maximize the benefits as provided by the business ecosystem of which a company is part.

References

1. Moore, J.F.: Predators and prey: a new ecology of competition. Harv. Bus. Rev. **71**(3), 75–86 (1993)
2. Iansiti, M., Levien, R.: Strategy as ecology. Harv. Bus. Rev. **82**(9), 69–78 (2004)
3. Fogelström, N.D., Gorschek, T., Svahnberg, M., Olsson, P.: The impact of agile principles on market-driven software product development. J. Softw. Maintenance Evol.: Res. Pract. **22**, 53–80 (2010)
4. Williams, L., Cockburn, A.: Agile software development: it's about feedback and change. Computer **36**(6), 39–43 (2003)
5. Olsson, H.H., Alahyari, H., Bosch, J.: Climbing the "Stairway to Heaven"—A multiple-case study exploring barriers in the transition from agile development towards continuous deployment of software. Software Engineering and Advanced Applications (SEAA), 38th EUROMICRO conference, pp. 392–399. IEEE Press (2012)

6. Bosch, J., Bosch-Sijtsema, P.: ESAO: A holistic ecosystem-driven analysis model. In: Proceedings of the 5th International Conference on Software Business (ICSOB), Cyprus, 15–18 June 2014
7. Sommerville, I.: Software Engineering, 6th edn. Pearson Education, Essex (2001)
8. Highsmith, J., Cockburn, A.: Agile software development: the business of innovation. Computer **34**(9), 120–127 (2001)
9. Humble, J., Farley, D.: Continuous Delivery: Reliable Software Releases through Build, Test and Deployment Automation. Addison-Wesley, Boston (2011)
10. Shalloway, A., Trott, J.R., Beaver, G.: Lean-Agile Software Development: Achieving Enterprise Agility. Addison-Wesley, Boston, MA (2009)
11. Bosch, J.: Building products as innovation experiment systems. In: Proceedings of the 3rd International Conference on Software Business (ICSOB), MIT, Cambridge, 18–20 June 2012
12. Hofmeister, C., Kruchten, P., Nord, R.L., Obbink, H., Ran, A., America, P.: A general model of software architecture design derived from five industrial approaches. J. Syst. Softw. **80**(1), 106–126 (2007)
13. Van der Linden, F., Bosch, J., Kamsteries, E., Kansala, K., Obbink, H.: Software product family evaluation. In: 3rd International Conference of Software Product Lines, SPLC 2004, pp. 110–129. Springer, Boston (2004)
14. Walsham, G.: Interpretive case studies in IS research: nature and method. Eur. J. Inf. Syst. **4**, 74–81 (1995)
15. Runesson, P., Höst, M.: Guidelines for conducting and reporting case study research in software engineering. Empir. Softw. Eng. **14**, 131–164 (2009)
16. Larman, C., Vodde, B.: Scaling Lean and Agile Development: Thinking and Organizational Tools for Large-Scale Scrum. Pearson Education, Boston (2009)

Chapter 3
Academia–Industry Collaboration: Getting Closer is the Key!

Anna Sandberg

Abstract Our greatest developers in industry do not typically view top researchers in academia as people who they can learn particularly much from, and our top researchers in academia do not typically view top developers as people who can contribute much to their research (except supply them with raw data). While researchers have spent years on post-university studies to learn the profession of data collection, data analysis, and data sensemaking, developers are trained to produce and deliver in time based on current best thinking. Taking different paths after the university studies soon ends up in different governing variables for developers and researchers of what brings value. This is a well-known phenomenon, and numerous articles and books are describing why this is the case (Barnes et al., Eur Manag J 20(3):272–285, 2002; Mathiassen, Inf Technol People 14 (4):321–345, 2002; Mora-Valentin et al., Res Policy 33(1):17–40, 2004; Gorschek et al., IEEE Softw 23(6):88–95, 2006; Rombach and Achatz, Research collaboration between academia and industry. In: Proc. Future of Software Eng. (FOSE 07), IEEE CS Press, pp. 29–36, 2007; Van den Ven, Engaged Scholarship: A Guide for Organizational and Social Research. Oxford Univ. Press, 2007). In the Software Center we have found practical ways on how to make developers and researchers appreciate the same values and by that join forces to solve complex software engineering issues. A key instrument is to "get close" on all levels from steering groups to reference groups and research teams. In the following chapter, we describe how we in the Software Center work in practice to stimulate a close collaboration and what is required to make this work over time.

3.1 Learning from Pre-software Center Collaborations

The telecom company Ericsson has collaborated with academia since the early 2000s. It started up as a local initiative between the two partners in the city of Gothenburg, Sweden, where a few researchers and industry practitioners came

A. Sandberg (✉)
Ericsson, Gothenburg, Sweden
e-mail: anna.sandberg@ericsson.com

J. Bosch (ed.), *Continuous Software Engineering*,
DOI 10.1007/978-3-319-11283-1_3

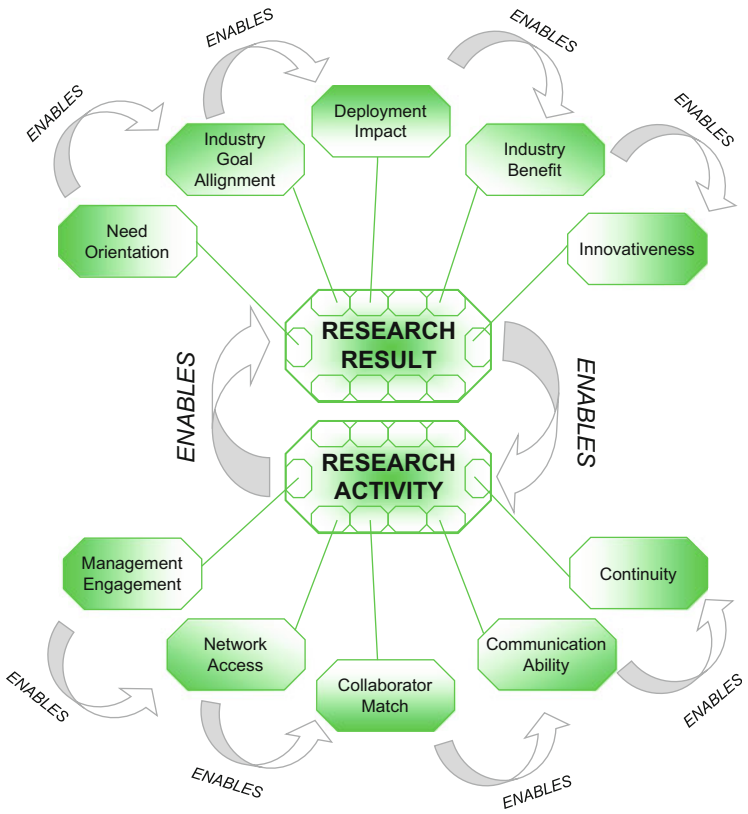


Fig. 3.1 Success factor model for academia–industry collaboration

together to join forces on challenging software engineering topics. This local initiative became a strategic global initiative between the two partners in 2006. In 2010, three of the involved researchers and industry practitioners came together and started to deeply analyze to understand what was making this collaboration working and not. This resulted in a model explaining the dependencies of ten different collaboration success factors and an IEEE Software publication [7]. Figure 3.1 describes that paying equal attention to the research activity and the research result enables in the end innovativeness and continuity of the research topic.

When the Software Center was started in 2011, two partners now became six (four industry partners and two academia partners). Two years later, another four industry partners had joined and several new researchers joined the center (and a few left). All these changes during a rather short time period have made it challenging to institutionalize what was learned from prior collaborations and deploy these. However, as this center's vision is about "10 x productivity in 10 years for the Nordic software industry," there has been a constant focus from the Software Center leads to guard and further emphasize factors making the including partners more successful. Doing so, the first phase of the Software Center

(2011–21013) has further contributed to the body of knowledge on successful collaboration, and it can all be summarized into one short statement: getting closer is key!

3.2 Getting Closer: On All Levels

Most studies about what makes change successful shows that management commitment is always in the top three of identified success factors [8, 9]. Viewing the Software Center as a change when it comes to getting academia and industry work closer, the learning about management commitment is indeed highly valid. Management commitment must start from the top, and the Software Center has a steering group consisting of representatives from all partners, both from industry and academia. This group has the overall responsibility to jointly decide on what the Software Center should focus on (e.g., research themes) and the forms for how to run its activities (e.g., type and length of the research). Everything is described in a contract where all partners engage to make sure we have a joint and solid collaboration foundation. The steering group has a task force group with representatives from all partners, both from industry and academia. This group has the responsibility to jointly dig deeper into the "what" decided by the steering group (e.g., research topics and questions within the research themes) and also propose on the "how" in more details (e.g., project proposals with collaborating partners and names). Research teams are then organized with partners from both industry and academia and started after a steering group decision. While the steering group and task force is fairly stable over time, the research teams can vary depending on how different partners choose to engage in different projects (Fig. 3.2).

Key to success at each level is of course depending on the software engineering competence and the effort all partners are willing to put into the different collaboration levels. Competence and effort combined with having the industry and academia partners working close help to view the software engineering challenges with the same goggles as well as strive towards reaching the same results. Academia sees the value of doing research on topics truly challenging for industry, while industry sees the value of the thorough way to understand and make sense of the challenges. The collaborating partners, at each level, start to respect each other's strengths, and over time their governing principles grow together. One example of getting the same governing principles is how the Software Center today views the publishing of papers. For academia this is the way to make a career, but for industry this has become a way of making the knowledge formally accepted and last over time. Another example is related to the deployment of the research findings. For industry this is everything as this is the way to improve practice, but for academia this has become a way to get reflections on the research results and inspiration for new research directions and topics. Again, what is required is to get the partners working close.

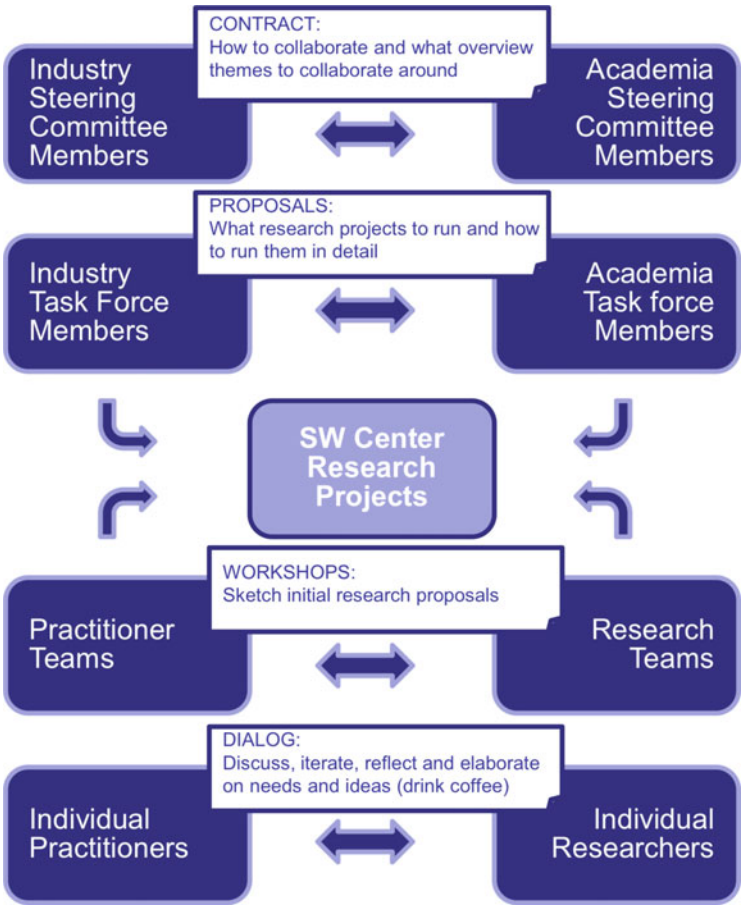


Fig. 3.2 Getting close—at all levels

3.3 Embracing Identified Success Factors

The Software Center is still in its cradle when it comes to leverage on its potential. The way forward is to continuously embrace identified success factors [7] and make them happen in practice. Sandberg et al. developed ten action principles to facilitate academia–industry collaborations in general, and below we describe how the Software Center embraces these to continuously improve its collaboration forms.

3.3.1 Address Activities to Ensure Results

Paying attention to both the research activity and the research result enables each other. The Software Center pays attention to management engagement within both industry and academia and the access to skilled researchers and practitioners as well as actual industry needs and goals and deployment impact. Once there is skilled people coming together to solve a true need, the foundation for successful research collaboration is in place.

3.3.2 Ensure Management Engagement

A structured setup exists (type of meetings, participants, meetings dates well ahead, well-prepared meeting agendas, etc.), which helps busy managers to engage in the activity that is typically not their normal full-time item to handle. The Software Center has defined a steering committee and a task force, where managers and leaders meet to discuss prepared agendas. In today's busy working environment, a meeting in outlook well ahead in time and an email with an agenda well ahead in time are two key enablers to have busy managers coming together at the same time to discuss the same topics.

3.3.3 Embrace Research Negotiations

Industry and academia lean on different governing variables. Therefore, it is vital that the industry and academia organize joint work which satisfies both parts. The research negotiation should therefore be embraced rather than viewed as something cumbersome. The Software Center has the goal to do world-class research in areas which helps improve the software industry productivity. Some brilliant research ideas are rejected if not being of enough relevance for industry, while some industry challenges are rejected if not contributing to be a frontline and world-class research.

3.3.4 Organize Get-Togethers

Opportunities for skilled engineers and researchers to meet and communicate are crucial for facilitating productive matches between researchers and practitioners. Normally, they work in different environments where they seldom meet, and when they meet, they do not necessarily appreciate each other's strengths. The Software Center organizes opportunities for the industry to come and present their key challenges to the researchers, who then based on their research interest and skills

come together in teams and start proposing research topics and projects for the industry. The Software Center also organizes these kinds of meeting on a more regular basis when the task force come together to evaluate propose research projects for the steering committee.

3.3.5 Fund Small Research Projects

Industry should distribute funding in small portions to keep research projects well aligned with its needs and goals. This approach also supports the frequent course changes needed to be agile. The Software Center organizes their research in 6 months sprints, where a project proposal should be able to deliver results each sprint. The results are not expected to have immediate industry impact, but the result should clearly show the path towards that.

3.3.6 Communicate Both Progress and Result

Research projects normally do not have immediate deployment impact, so maintaining organizational interest in them requires finding ways to visualize progress as well as results. Within the Software Center, we today have at least three different research topics that have stayed current for more than 3 years having new project proposals on the same topic in the same theme. These research teams have managed to communicate and visualize their progress in a way to attract the steering committee to decide on continuation.

3.3.7 Attend to Both Needs and Goals

Research projects that target an industry need and involve a dedicated business unit with an immediate goal related to this need are more likely to succeed than those that address only general needs. The Software Center assures every research project to have dedicated industry representatives willing to engage and spend time in the research project. This way the research project becomes equipped with practitioners who have real problems to solve now (otherwise, they will not be willing to spend particularly much time on it).

3.3.8 Be Agile

Just as in other industrial projects, research projects must deal with an ever-changing business environment. To facilitate results that have deployment impact, projects must accept and respect that industry goals and research directives change. The Software Center runs its research projects in relatively short sprints (6 months), which allows for changing direction for a research topic without interrupting the ongoing activities. One researcher explained this as "waltzing with industry to be more like a quick-step dance," but once he embraced this fact, he did very well with his research projects.

3.3.9 Allow Innovation to Emerge from Needs

High attention to both deployment impact and industry benefits facilitates innovativeness. Research that reflects on industry practice, working closely with its prioritized needs and goals, is a recipe for success. The Software Center has the industry problem and its solutions as priority one. In this way, researchers start with a problem in need of innovativeness to have its problem solved. Such a start will in itself facilitate innovativeness.

3.3.10 Realize that Collaborative Research Involves Learning

It takes time for both parties to understand each other's needs. Researchers must learn to be agile towards industry needs, and practitioners must learn to appreciate research rigor. Such learning requires time, perseverance, and conscious reflection. The Software Center has via a number of internal meetings started to reflect on how research is best executed in their context.

3.4 The Next Step Towards Increased Collaboration

Getting closer and appreciating each other's strengths are a precondition for having successful academia–industry collaborations. When getting closer the involved partners come to understand what brings value at each side and how these values can be combined in joint research projects. Paying attention to identified success factors improves the collaboration in general and more specifically appreciating the joint work and its outcome. Moving forward, the Software Center will need to continuously nurse and understand more about the factors that make them

successful. The Software Center is fully committed to do so, but the practical methods to continuously come together and embrace the collaboration factors need to be even more prioritized and further fine-tuned.

References

1. Barnes, T., Pashby, I., Gibbons, A.: Effective university–industry interaction: a multi-case evaluation of collaborative R&D projects. Eur. Manag. J. **20**(3), 272–285 (2002)
2. Mathiassen, L.: Collaborative practice research. Inf. Technol. People **14**(4), 321–345 (2002)
3. Mora-Valentin, E.M., Montoro-Sanchez, A., Guerras-Martin, L.A.: Determining factors in the success of R&D cooperative agreements between firms and research organizations. Res. Policy **33**(1), 17–40 (2004)
4. Gorschek, T., Wohlin, C., Carre, P., Larsson, S.: A model for technology transfer in practice. IEEE Softw. **23**(6), 88–95 (2006)
5. Rombach, D.H., Achatz, R.E.: Research collaboration between academia and industry. In: Proc. Future of Software Eng. (FOSE 07), IEEE CS Press, Washington, pp. 29–36 (2007)
6. Van den Ven, A.: Engaged Scholarship: A Guide for Organizational and Social Research. Oxford University Press, Oxford (2007)
7. Stelzer, D., Mellis, W.: Success factors for organizational change in software process improvement. Softw. Process Improv. Pract. **4**(4), 227–250 (1998)
8. Grady, R.B.: Successful Software Process Improvement. Prentice Hall, Upper Saddle River (1997)
9. Stelzer, D., Mellis, W.: Success factors of organizational change in software process improvement. Softw. Process – Improv. Pract. **4**(4), 227–250 (1998)

Part II
Agile Practices

This part discusses the second step on the Stairway to Heaven, i.e., the adoption of agile development practices. There are four chapters in this part. The first chapter is concerned with the role of the software architect in a large-scale agile development organization. The chapter introduces a set of architect roles and a set of activities that need to be performed by these different architect roles. The second chapter studies the inhibitors to speed that agile teams experience because of the dependencies on both other agile teams and other teams and departments in other functions in the organization. The subsequent chapter uses the problem analysis in the aforementioned chapter to present a validated framework, including a set of practices, to optimize for different types of speed in product development. The final chapter in this part is concerned with improving customer responsiveness in large-scale development organizations. The approach employs customer-specific teams where selected agile teams work with prioritized customers while merging the solutions requested by their customer into the main product code base. The approach balances the need for scale, i.e., getting many features out in the general product, with the need for responsiveness to prioritized customers.

Chapter 4
Role of Architects in Agile Organizations

Antonio Martini, Lars Pareto, and Jan Bosch

Abstract Agile software development is broadly adopted in industry and works well for small-scale projects. In the context of large-scale development, however, there is a need for additional structure in the form of roles and practices, especially in the area of software architecture. In this chapter, we introduce the CAFFEA framework that defines a model for architecture governance. The framework defines three roles, i.e., chief architect, governance architect, and team architect, as well as a set of practices and responsibilities assigned to these roles. The CAFFEA framework has been developed and validated in close collaboration with several companies.

4.1 Background

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. Support for such goals is given by Agile software development (ASD), which has been proven successful in small software projects in the last decade [1].

Some frameworks including sets of practices have been proposed for implementing Agile principles of which few are suitable for large projects (e.g., Scrum). However, a major gap in such frameworks is the lack of activities to enhance agility in the task of developing and maintaining a reference architecture, necessary for the development of portfolios of products sharing large amount of software (e.g., platforms) [1, 2]. This lack might lead to an underdeveloped or quickly eroded architecture, which is needed to coordinate large software projects, to enhance future development (future features), and to assure cost reduction (through software reuse). The key need, for the companies, is to reach the ability of Agile architecting and therefore to redefine the role of the software architects in an Agile organization, which means understanding which activities related to

A. Martini (✉) • L. Pareto • J. Bosch
Chalmers University of Technology, Gothenburg University, Gothenburg, Sweden
e-mail: antonio.martini@chalmers.se; Jan@JanBosch.com

J. Bosch (ed.), *Continuous Software Engineering*,
DOI 10.1007/978-3-319-11283-1_4

software architecture are necessary to be carried out and by whom. We have therefore redefined the key roles for architects, with the responsibilities of architecture management that were missed in large Agile organizations: chief architect, governance architect, and team architect.

- Chief architects, responsible for the whole overall portfolio architecture, which might include more products and more than one system.
- Governance architects, responsible for areas of the architecture, related to single products or systems or sub-systems, but not related to only one team.
- Team architects, the usually most experienced developer in a team who have the most knowledge about the architecture and support/lead the team on such area.

Different levels in the hierarchy are usually connected with the level of abstraction of the software architecture and design for which the architects are accountable.

Such generic architecture roles might change from company to company, but important questions are: What are they actually doing? What should they do? How should they spend their time?

There are many activities suggested by the research community that need to be done to develop and maintain a sound software architecture [3–12]. However, little is said about who is responsible for which activities in a large Agile software development organization. This gap is especially important where ASD is employed, and the organization tends to avoid big time investments in upfront design in favor of a shorter time to market.

In our previous studies, we have shown that the Agile teams need to communicate and coordinate, and the need for strategic input is of utmost importance [13, 14]. Also, in [15] the authors introduce Agile in an SPL setting but report an open issue as:

> the management of architecture evolution and refactoring without sacrificing the principles of agility and self-managed teams.

Leffingwell [16] introduced the concept of *architecture runway* as an

> existing or planned infrastructure sufficient to allow incorporation of current and anticipate requirements without excessive refactoring.

We will use such concept when we will explain the need for a Runway Team, which in [16] has been identified as a Prototyping Team. Leffingwell also mentions the possibility, for large systems, to have architects outside of the teams. However, no specific roles and activities have been mentioned for the architects: on the contrary, Leffingwell advocates the current impossibility of placing traditional architect roles (e.g., system architects) present in many organizations (especially in products with embedded software) into Agile frameworks. We have therefore collected roles with specific responsibilities in a framework called CAFFEA (continuous architecture framework for embedded software and Agile).

4.2 Research Design

We planned a multiple-case embedded case study involving seven sites in five large software development companies. For confidentiality reasons, we will call the companies A, B, C, D, and E. The main rationale for selecting the cases was that they needed to be developing software product lines and they had adopted or at least been in transition to ASD.

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

4.2.1 Cases Description

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

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

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

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

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

All the companies have one or more product lines and have adopted a component-based software architecture, where they reuse a substantial part of the system, such as several components or a shared platform. The language that is mainly used is C and C++, with some parts of the system developed in Java and Python. Company A uses a domain-specific language (DSL) to generate C code, while company E uses a DSL for specifying rules to be converted into libraries. The development at C_3 involves extensive XML.

4.2.2 *Data Collection and Analysis*

First we conducted a literature review to find the main state-of-the-art architecture activities that need to be done to develop and maintain a sound software architecture [3–12]. During the interactive workshops, we mapped such activities into roles with the help of the informants. With such process we found which activities were already covered by the Agile process employed at the companies and which ones were not. The workshops were recorded and transcribed. The analysis was done following an approach based on grounded theory [17], mixing inductive and deductive techniques and using a tool for qualitative analysis, to keep track of the links between the codes and the quotations they were grounded to. The overall process is shown in Fig. 4.1.

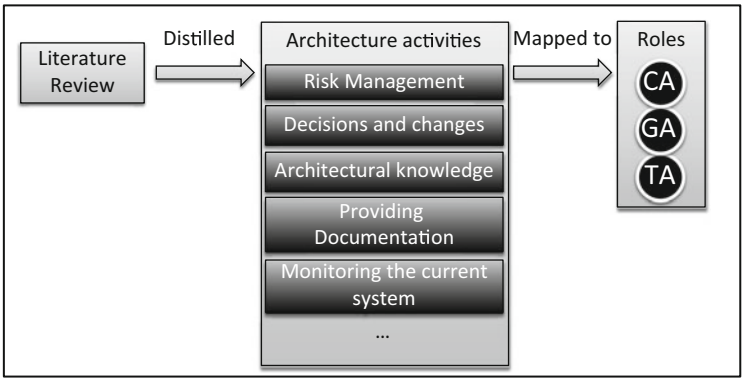


Fig. 4.1 CAFFEA framework and its components. The *arrows* show the analysis steps leading to the results

4.3 Roles and Activities

4.3.1 Chief Architect

The main role of the Chief Architect (CA) is to take high-level decisions and to drive and support governance architects and teams in order to reach strategic goals such as the development and maintenance of a sound architecture that would support the business goals of the organization. In many cases such goals include the support for multiple products or product lines, which assures multiple incoming to the company. All of the cases that we have analyzed have such portfolio of multiple customers. The main activities for the CA are described in the following, grouped by the main areas.

4.3.1.1 Risk Management

The CA should drive the activities connected to risk management related to product and architecture evolution. Such an activity is done usually on a 2–3-year scope.

The CA is usually not directly involved in the detailed development; however, in order to take feasibility decisions, the CA needs to elicit the information about the current status of the system from the governance architects and team architects by the institution of a virtual architecture team. The presence of CAs in estimation is usual, but the involvement of the GA and TA is not often applied, although many informants reported that such roles are important for understanding the feasibility of deliveries that require commitment. The virtual team should also involve Product Owners since risks in terms of cost related to the architecture need to be matched with the business model and goals of the company.

4.3.1.2 Managing Decisions and Changes

The CA takes decisions that affect more products, for example, tools and frameworks (e.g., libraries) that need to be used cross-company, or high-level design decisions (e.g., architectural styles, archetypes). Such decisions cannot be taken by development teams alone but need to be taken by architects with high experience and overall picture of the system.

Decisions on the adoption of frameworks, tools, and high-level design need to be followed by decisions to support them by educating the development teams. Such decisions involve the budget, so they need to be taken by (or in combination with) product management.

4.3.1.3 Pattern Distillation

The chief architect should participate in the learning activity with the governance architects (described later) on distilling patterns. Such retrospective activity is not done per feature, but it involves the architectural analysis of the system for a whole product or product line. It's not a frequent activity, but it's usually performed before high-level patterns are chosen and product-wise (and therefore cost-impacting) decisions need to be taken, for example, the starting of a complete new product or its inclusion in the organization portfolio.

4.3.1.4 Providing Architecture Documentation (Communication Output)

One of the main tasks for the CA should be to communicate the overall architecture and the purpose of it. The best way, according to the informants, is to broadcast such information in a visual way and in person. The mentioned means for this activity are videos, "road shows," and plenary sessions with all the consumers of the architecture.

The CA needs to "order" education for the consumers after a decision is made to change infrastructures, high-level design patterns, or processes. It's not likely that the CA alone is able to take care of such education. For what concerns the architecture, the CA should delegate most of the documentation creation and education to the GAs.

4.3.1.5 Receiving Input About the Current Status of the System (Communication Input)

According to the informants, the current communication practices lack good mechanisms for providing input to the CAs. In fact, CAs cannot go around the organization and talk to every consumer of the architecture. Input is fundamental for the CA in order to take informed decision and to be aware of bad decisions taken previously that need to be addressed (e.g., about tool chains not working as expected). Some of the means for such communication input are:

- Plenary sessions
- Questionnaires
- Forums

4.3.2 Governance Architect

The Governance Architect (GA) is an intermediate role in the organization and the key for scalability of the architecture development and maintenance in a large Agile setting. Such role should function as a link among several teams that need to be coordinated and supported when developing features within a (evolving) system architecture. The main activities for the CA are described in the following, grouped by the main areas.

4.3.2.1 Inter-features Architecting (Architecture Decision)

The GA should be responsible for decisions about inter-feature architecting. The development of certain features might require adding design patterns or making sure that the architecture is optimal not just for one feature but for more, for example, to support features planned for the near future. This is something that requires a wider view of the system architecture than the view focused on a single feature typical of the Agile team. These decisions are usually taken considering also high-level architecture. An example of such architecture is the production of a meta-model for the understanding of the system components and their connectors (relationships).

Besides the technical integration of the features, the GA needs to monitor and drive an architecture that would minimize or make clear the inter-team effects or the development of each team on the other. For this purpose, there is also the need to define the boundaries for the teams, i.e., the critical points in the architecture in which the risk is high for the team to interact with the others (see also documentation [*]). GAs need to take architecture decisions involving at least 2 or more teams, although the more teams are involved, the more coordination effort needs to be spent on such activities.

The GA needs to take product-oriented decisions, but also to monitor and support the team in following such decisions. With the focus of the team on the features, someone needs to be able to "protect" the whole product, for example, in terms of quality or pattern distillation (see CA [*]). In order to have such a mind-set and information, there is the need to maintain a strong and iterative communication link with the CA.

4.3.2.2 Architecting for Testability

An important point made by the informants is the need to architect to achieve testability. The continuous integration team, responsible of the integration at (sub-) system level, is a stakeholder of the team. Decisions about patterns and measurement mechanisms to be embedded in the code for testing and quality management need to be taken by the GAs.

4.3.2.3 Risk Management

One of the main needs in a large organization is to balance the prioritization of short-term and long-term goals. Such risk management activity is done on different levels, and the GA should be a connector between the levels by participating in the relative risk management activities. The GAs need to plan long-term product development and architecture evolution with CAs and Product management (High-level Product Owners), while they also need to be involved in the prioritization of features and architecture improvements at lower levels (closer to design and implementation) with the TA and, if present, a lower-level Product Owner. In two of the cases studied, belonging to the same large organization but being quite disconnected sites, there was the presence of a so-called Operational Product Owner, who was responsible for the feature prioritization directly with the team.

4.3.2.4 Controlling Erosion

The GAs are the main drivers for monitoring and reacting to architecting erosion. Such activity is quite complex, and the shift to Agile has also led to a shift from a document-centric architecture monitor to the more iterative and in-person participation of the architects. Controlling erosion needs also the support within the team by the TAs, but the information provided by the team lead to decisions that have to be taken with large perspective (see [*] decisions). On the other hand, the CA cannot monitor the whole system to understand what is happening (see [*] CA controlling erosion).

Consequently, the GAs are needed to lead architecture erosion control by coordinating sub-activities, such as:

- Retrospective sessions: the topic of the retrospective sessions (e.g., postmortem analysis) suggested by Agile practices is team performance. From such activity, the team learns how to be more efficient. However, specific sessions or sub-sessions need to be organized and led by the GAs, in order to focus also the retrospective towards architecture. Such sessions might be organized at the end of:
  - Sprint
  - Project
- Code reviews: a usual practice is to peer-review the code. However, such practice seems to be quite a costly one and is not always possible because of the time constraints due to the low number of GAs present in the organization compared to the amount of code and the time invested in the other activities listed also in this section. For this reason, the GA should be able to order automation for such code-reviews.
- Order automation: to facilitate the monitor of erosion, the inconsistencies with respect to the desired architecture or architecture principles and requirements

need to be visualized by analyzing the system. The GAs should be able to "order" (allocate the budget and resources) automation to an infrastructure team (or third party vendor) that could put in place a tool for visualizing and monitoring some erosion.

4.3.2.5 Architecture Education for the Teams (Communication Output)

Although the chief architect is responsible for the production and communication of high-level architecture guidelines and decisions, the governance architect should be responsible for the capillary spreading of knowledge to all the teams. In such role, the governance architect, in the organization, needs to have enough overview and architecture knowledge to express the important concepts.

The main means for communicating architecture and educating the teams is maintaining architecture documentation (inter-team, inter-feature): although the transition to ASD has questioned the need for the maintenance of superfluous architecture documentation, part of it is critical and needs to be very well created and kept updated. Such documentation is the one concerning architectural patterns and requirements that are shared by more than one team. As an example, there might be patterns that involve temporal requirements (e.g., involving the access to a centralized database) that need to be expressed (and verified) in order to keep the teams from hindering each other. An important requirement for the architecture documentation is that it should be well navigable by the team members, meaning that the link between the documents and the actual source code should be well maintained.

4.3.2.6 Knowledge of the Status of the System (Communication Input)

Another important point in architecture communication is that the architect should be able to have knowledge of the status of (part of) the system. The main problem that would arise without having such knowledge would be to have architects who define and evolve a system architecture that is completely different from what is actually implemented. Such situation obviously hinders the decisions when risk analysis is done for the augmentation of the products, e.g., the implementation of new features. It is of utmost importance, therefore, that the architects continuously and iteratively check the actual status of the system. For doing so, there might be tools, but, according also to the Agile view of the importance of face-to-face communication, architects should organize and participate in events including experienced members of the teams. Such events are in line with the continuous learning principle and with the practice, suggested in some Agile frameworks, of conducting retrospective. In order for the architecture to be correctly communicated to the governance architect, architecture retrospectives (i.e., retrospective sessions

focused on reflecting on the current status of the architecture) have been found to be a good means.

4.3.3 Team Architect

The Team Architect (TA) is the actual executor of the architecture in the FT. The role is important for the capillary spreading of architecture knowledge and for collecting input from the FT. However, this role is not a dedicated role in the team but rather a set of responsibilities that need to be mapped to a member of the team, who might change from time to time.

4.3.3.1 Risk Management

As mentioned in the previous sections, the TA should participate in risk management activities with the CA, GA, and P in order to represent the interest of the teams in feasibility discussions.

4.3.3.2 Managing Decisions and Changes

The TA leads detailed architecture decisions on a detailed design level.

4.3.3.3 Providing Architecture Documentation (Communication Output)

The TA takes input from the GA and should be able to communicate the architecture to support the FT education packages created by the GAs.

4.3.3.4 Monitoring the Current Status of the System (Communication Input)

The TA is responsible for collecting FT's input, to be communicated to the GA. This way, the TA lifts decisions for architecture evolution and decisions that might affect other FTs.

4.3.4 Gap in the Current Practices

The main groups of practices that have been recognized as currently missing are:

- Architecture risk management (prioritization of short-term and long-term tasks)
- Architecture decision and changes
- Communication of architecture, composed by two-way directions:
  - Providing architecture directions
  - Monitoring 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 [18, 19] and is concerned with the organizations taking risk-informed architecture decisions about which architecture changes, such as refactoring or evolution (considered the actual debt if not done) need to be conducted for having an acceptable ratio of investment/effort. The lack of architecture technical debt management might quickly lead the companies to crisis points where adding new business value to the software product lines (new features or new products) incur in major efforts, paralyzing the long-term responsiveness [20].

Conclusion

Agile software development is broadly adopted in industry and works well for small-scale projects. In the context of large-scale development, however, there is a need for additional structure in the form of roles and practices, especially in the area of software architecture.

In this chapter, we introduced the CAFFEA framework that defines a model for architecture governance. The framework defines three roles, i.e., chief architect, governance architect, and team architect, as well as a set of practices and responsibilities assigned to these roles. For the chief architect, these practices include risk management, managing decisions and changes, pattern distillation, providing architecture documentation, and receiving input about the current status of the system. The governance architect is concerned with inter-features architecting (architecture decisions), architecting for testability, risk management, controlling erosion, architecture education for the teams, and maintaining knowledge of the status of the system. Finally, the team architect focuses on risk management, managing decisions and changes, providing architecture documentation, and monitoring the current status of the system.

The CAFFEA framework has been developed and validated in close collaboration with several companies. Currently, most of the companies in the Software Center have adopted or are in the process of adopting the CAFFEA framework.

Although the framework has brought significant benefit for the companies, there still are some open items, as presented earlier in the chapter, and in future work, we aim to address those as well as validate the framework with other companies in other industries.

References

1. Dingsøyr, T., Nerur, S., Balijepally, V., Moe, N.B.: A decade of agile methodologies: towards explaining agile software development. J. Syst. Softw. **85**(6), 1213–1221 (2012)
2. Daneva, M., van der Veen, E., Amrit, C., Ghaisas, S., Sikkel, K., Kumar, R., Ajmeri, N., Ramteerthkar, U., Wieringa, R.: Agile requirements prioritization in large-scale outsourced system projects: an empirical study. J. Syst. Softw. **86**(5), 1333–1353 (2013)
3. Kruchten, P.: What do software architects really do? J. Syst. Softw. **81**(12), 2413–2416 (2008)
4. Tang, A., Avgeriou, P., Jansen, A., Capilla, R., Ali Babar, M.: A comparative study of architecture knowledge management tools. J. Syst. Softw. **83**(3), 352–370 (2010)
5. Pareto, L., Eriksson, P., Ehnebom, S.: Architectural descriptions as boundary objects in system and design work. Model Driven Eng. Lang. Syst. **6395**, 406–419 (2010)
6. Williams, B.J., Carver, J.C.: Characterizing software architecture changes: a systematic review. Inf. Softw. Technol. **52**(1), 31–51 (2010)
7. de Silva, L., Balasubramaniam, D.: Controlling software architecture erosion: a survey. J. Syst. Softw. **85**(1), 132–151 (2012)
8. Qumer, A.: Defining an integrated agile governance for large agile software development environments. In: Concas, G., Damiani, E., Scotto, M., Succi, G. (eds.) Agile Processes in Software Engineering and Extreme Programming, pp. 157–160. Springer, Berlin, Heidelberg (2007)
9. Drury, M., Conboy, K., Power, K.: Obstacles to decision making in Agile software development teams. J. Syst. Softw. **85**(6), 1239–1254 (2012)
10. Zimmermann, O., Miksovic, C., Küster, J.M.: Reference architecture, metamodel, and modeling principles for architectural knowledge management in information technology services. J. Syst. Softw. **85**(9), 2014–2033 (2012)
11. Unphon, H., Dittrich, Y.: Software architecture awareness in long-term software product evolution. J. Syst. Softw. **83**(11), 2211–2226 (2010)
12. McAvoy, J., Butler, T.: The impact of the Abilene Paradox on double-loop learning in an agile team. Inf. Softw. Technol. **49**(6), 552–563 (2007)
13. Martini, A., Pareto, L., Bosch, J.: Enablers and inhibitors for speed with reuse. In: Proceedings of the 16th International Software Product Line Conference, vol. 1, pp. 116–125. New York, USA (2012)
14. Martini, A., Pareto, L., Bosch, J.: Communication factors for speed and reuse in large-scale agile software development. In: Proceedings of the 17th International Software Product Line Conference, pp. 42–51. New York, USA, (2013)
15. Bosch, J., Bosch-Sijtsema, P.M.: Introducing agile customer-centered development in a legacy software product line. Softw. Pract. Exp. **41**(8), 871–882 (2011)
16. Leffingwell, D.: Scaling Software Agility: Best Practices for Large Enterprises. Pearson Education (2007)
17. Strauss A., Corbin, J.M.: Grounded Theory in Practice. SAGE (1997)
18. Tom, E., Aurum, A., Vidgen, R.: An exploration of technical debt. J. Syst. Softw. **86**(6), 1498–1516 (2013)
19. Nord, R.L., Ozkaya, I., Kruchten, P., Gonzalez-Rojas, M.: In search of a metric for managing architectural technical debt. In: 2012 Joint Working IEEE/IFIP Conference on Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), pp. 91–100 (2012)
20. Martini, A., Bosch, J., Chaudron, M.: Architecture technical debt: understanding causes and a qualitative model. Presented at 40th Euromicro Conference on Software Engineering and Advanced Applications, Verona, pp. 85–92 (2014). doi:10.1109/SEAA.2014.65

Chapter 5
Teams Interactions Hindering Short-Term and Long-Term Business Goals

Antonio Martini, Lars Pareto, and Jan Bosch

Abstract A known problem in large software companies is to balance the return on investment coming from short-term and long-term business goals dependent on the responsiveness of the companies. We have conducted an investigation on three large product line companies employing Agile software development (ASD) to better understand this problem: we have recognized several challenges that were hindering one or both kinds of business goals. Interaction challenges were quite critical among the Agile teams but also between the Agile team and other parts of the organization, such as architects and product management. We also further investigated which root factors were behind the interaction challenges, what symptoms can be recognized in the organization to spot the interaction challenges, and how they were related to the recent employment of ASD in the companies.

5.1 Background

Large software industries strive to make their development processes fast and more responsive, minimizing the time between the identification of a customer need and the delivery of a solution. The trend in the last decade has been the employment of Agile software development (ASD) [1]. At the same time, the responsiveness in the short-term deliveries should not lead to less responsiveness in the long run.

There are three relevant goals dependent on speed ([2], visible in Fig. 5.1): 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 requests to an existing product are realized (evolution speed).

Increasing the speed in each of these goals contributes to the return on investment of the companies. However, there might be challenges [3, 4] decreasing the speed (which in turn affects negatively the ROI of R&D). Such challenges need to be identified and mitigated through dedicated practices.

A. Martini (✉) • L. Pareto • J. Bosch
Chalmers University of Technology, Gothenburg University, Gothenburg, Sweden
e-mail: antonio.martini@chalmers.se; Jan@JanBosch.com

J. Bosch (ed.), *Continuous Software Engineering*,
DOI 10.1007/978-3-319-11283-1_5

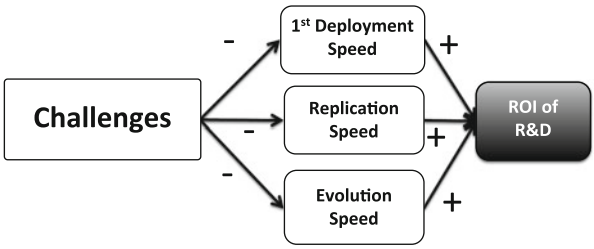


Fig. 5.1 Challenges in different kinds of speed may hinder the reaching of business goals and therefore return on investment

We investigated such challenges in three partners of the Software Center, described below as companies A, B, and C.

5.1.1 The Participating Companies

We investigated the previously mentioned challenges in 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.

Company A was involved in the automotive industry. Part of the development was done by suppliers, some by in-house teams following Scrum. The surrounding organization was following a stage-gate release model. Business was driven by 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. 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 have feature development roles. Some of the teams had special supporting roles (technology, knowledge, architecture, etc.). 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.

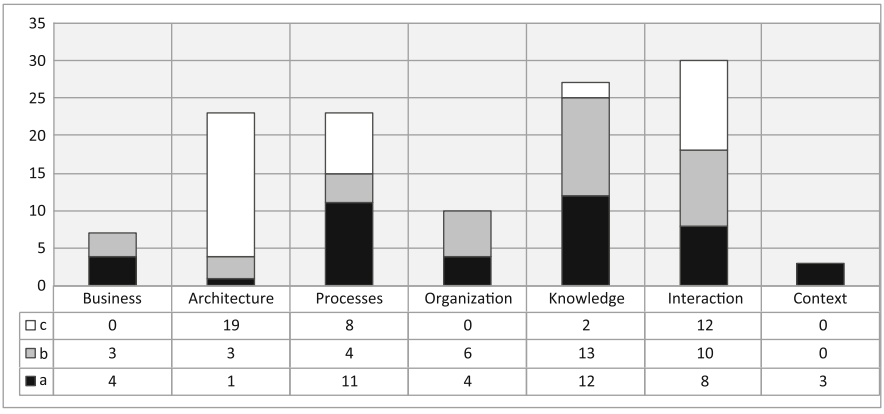
5.2 Challenges Hindering Business Goals Based on Speed

The first step in the investigation was to gather in-depth qualitative data from few cases in order to understand the challenges. We analyzed the data through the grounded theory approach, which is useful for understanding emerging factors and to create theories to model complex phenomena embedded in real contexts.

The qualitative investigation brought to light 114 challenges [2]. After a careful categorization is achieved through the application of grounded theory, we obtained a distribution of the challenges in several areas of improvement. Such distribution is showed in the bar charts in Fig. 5.1. These diagrams suggest which areas the informants were most concerned with in each case and among all.

The even distribution of the 30 challenges in the *Interaction* column of Fig. 5.2 (12, 10, 8, for cases A, B, and C, respectively) emphasizes that interaction is a common concern at all sites. More specifically, the detailed bar charts outline how interaction between teams involved with a common asset (e.g., when reusing a component or an entire platform) should be well supported, as should interaction across projects related to the same product. Good interaction infrastructure is also necessary among the development teams, between teams, and the units these depend on, e.g., systemization and business units. Without adequate interaction, reuse risks being unorganized and ineffective (ad hoc reuse), or it will cause a huge penalty in terms of speed.

The data from this study provided us with sufficient motivation for further studying intraorganizational interaction challenges. The category containing such challenges was the most populated one, which showed its importance, and it was the category with the most evenly distributed challenges among the studied sites, which suggested a high degree of generalizability of such 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

Fig. 5.2 Distribution of the challenges in cases and categories

5.3 Interaction Challenges in Depth: Which Parts of the Organization Do the Agile Teams Struggle to Interact With?

Through a quantitative investigation among 36 participants [5], we produced an ordered table of the 23 interaction challenges previously investigated, their recognition in terms of frequencies, and their perceived spread in the companies. Furthermore, we managed to highlight challenges recognized by the respondents in some contexts (studied companies) but not in others and to outline the ones that were recognized by some roles but not by others. The challenges are listed in Table 5.1 and the results are summarized in Fig. 5.3, which shows the overall recognition of the challenges, coded by Q_{01}–Q_{23} in the leftmost column.

Below we provide a more detailed description of the content in Fig. 5.3. The second column from the left (*Recognition*) displays the percentage of respondents (calculated on the valid answers provided) that have recognized that challenge. All the challenges 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), and a bold border groups the rows of the challenges included in each level. Nine challenges, with more than 75%, are strongly recognized. The challenges with more than 50%, other 11, are strongly recognized but controversial, which means that even if there are more "yes" than "no," there is a clear contrast in the answers. The last three challenges are weakly recognized but controversial, which means that the number of "no" is more than the number of "yes," but the percentage of the latter ones is close to the former ones. We don't have any weakly recognized challenge.

In the middle column (*Rank with Spread*) we have showed the spread calculated using the means of the answers weighted from 0 to 4: such results show, together with the recognition, how much the respondents perceive the spread of the challenge around them. This changes partially the order of the challenges, as we can see for Q14 and Q01: the former one has a higher recognition rate, but the latter one is evaluated to be more spread through the company, and the two aspects are summarized in the means. Notice that the spread doesn't influence the previous categorization of the challenges (strongly, weakly, etc.).

In the last two columns from the left (*Depends on Context* and *Depends on Role*), we have marked the challenges with an "x" when we have statistical evidences that the challenge has the property and with "!" when we have evidences that have to be further confirmed. Such results are further explained in the following sections.

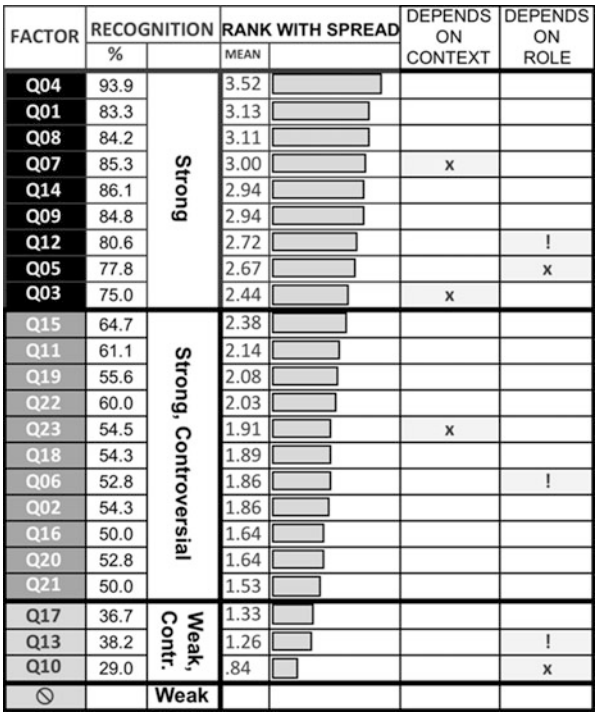
5.3.1 Challenges Related to Different Companies

The distribution of the answers for the challenges rarely changes with respect to the respondents' company. However, for three of them (Q03 Q07 Q23), we found a

Table 5.1 Challenges ordered by their recognition and spread, annotated if dependent on context or role

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 bugs or defects
Q04	Developers and system engineers are not co-located, which causes interaction 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 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 it cannot be guided properly
Q09	A development unit has to build a component to be integrated in other units' (projects') system, but the interaction with them is not sufficient
Q10	A development unit was forced to integrate a common component
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 interaction problems
Q13	Team's (or unit, project) lack of will to integrate a common component
Q14	Lack of understandable documentation or of proper interaction 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' mind-set 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 interaction
Q18	Different favorite tools in the same team create interaction 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 (e.g., 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
Q23	Loss of knowledge about a reused framework's variation points, for example, a framework created some years earlier

significant difference among the participants' responses. The identified challenges are reported in Table 5.1 with "x" on the column *Depends on Context*. In the following we describe the differences among the companies:



FACTOR	RECOGNITION %		RANK WITH SPREAD MEAN		DEPENDS ON CONTEXT	DEPENDS ON ROLE
Q04	93.9	Strong	3.52			
Q01	83.3		3.13			
Q08	84.2		3.11			
Q07	85.3		3.00		x	
Q14	86.1		2.94			
Q09	84.8		2.94			
Q12	80.6		2.72			!
Q05	77.8		2.67			x
Q03	75.0		2.44		x	
Q15	64.7	Strong, Controversial	2.38			
Q11	61.1		2.14			
Q19	55.6		2.08			
Q22	60.0		2.03			
Q23	54.5		1.91		x	
Q18	54.3		1.89			
Q06	52.8		1.86			!
Q02	54.3		1.86			
Q16	50.0		1.64			
Q20	52.8		1.64			
Q21	50.0		1.53			
Q17	36.7	Weak, Contr.	1.33			
Q13	38.2		1.26			!
Q10	29.0		.84			x
⊘		Weak				

Fig. 5.3 Challenges ordered by their recognition and spread, annotated if dependent on context or role

Q03: *Project-related bugs or defects*.
For challenge Q03 there is a great difference between companies B and C: the respondents in company C don't recognize the challenge, while the ones in company B strongly recognize it.

Q07: *There is no time to improve parts of software shared among projects*.
For challenge Q07 the respondents in company B strongly recognize it, while the other ones stand in the middle. This shows that this challenge is very present in company B while controversial in the other companies. With respect to Table 5.1, 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*.
Company B recognizes this challenge more than company C.

5.3.2 Different Roles' View on Some Challenges

We found that only few of the answers for the challenges changed when given by different respondents with different roles. These are marked with "x" in Fig. 5.3 in the column *Depends on Roles*.

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 (e.g., time or workload).*
For this challenge, managers and system engineers show 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.*
Some results from the ANOVA tests have highlighted interesting differences among the means of the respondents on the following challenges. These evidences, though, have to be read with care: they can be considered hints for further research more than strong evidences. For this reason we have marked these challenges with "!" in Fig. 5.3.

Q06: *Evaluation of costs and feasibility of new features don't take into account implementation issues and constraints, causing inaccuracy in budget and resources (e.g., time or workload) allocation.*
The recognition of this challenge changed greatly between system engineers and designers, while the other roles stand in the middle.

Q12: *Different attitudes and values of distributed (not co-located) teams (or units, projects) caused communication problems (tensions, synchronization problems, overheads, delays, misunderstandings, etc.).*
For this challenge, some designers seem to not recognize the challenge, while the other roles show agreement in its recognition.

Q13: *Team's (or unit, project) lack of will to integrate a common component (a component shared with other development units).*
For this challenge, designers and testers don't recognize the challenge, while managers and especially system engineers have experienced the problem (even if there are controversial opinions).

5.3.3 Prioritization of Interaction: Selection of Critical Organizational Boundaries in Need for Intergroup Interaction Improvement

From the challenges and their prioritization, it's possible to identify which intergroup boundaries needed attention for increasing or decreasing interaction (depending on the challenge/boundaries, we wanted to maximize or minimize interaction through selected practices).

The strongly recognized challenges highlight different interaction problems affecting the interfaces between the development team and other parts of the organization. ASD requires frequent feedback in order to keep the development

iterative. These results show that in large organizations, there is the need to improve the following interfaces between Agile teams and other organizational entities:

- *Team—System engineers*: Q04 is the most recognized challenge, almost by all the respondents (93%, which include many system engineers): system requirements should be continuously discussed between system engineers and the Agile team. Software reuse has to be recognized at a system level to be implemented also in software components. According to the qualitative data, (partial) co-location between system engineers and the Agile team would mitigate or solve this challenge: the Agile team needs explanation and agreement on the requirements, for example, in understanding which ones are fixed and which ones are not (and could be changed by system engineers to ease software development). Since we have studied only companies producing embedded systems, we can infer that this challenge is generalizable in this context, but we can't generalize it for pure software developing companies.
- *Team—Product development and management*: the respondents' answers about challenges Q01 and Q02 (processes-related interaction) contrast visibly. It seems that ASD suffers particularly when it has to interface to the product development rather than with the project one. 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 might be bound to the surrounded product development process, which hinders the quickness that would be gained by employing ASD. However, we cannot say if this is a real constraint or a challenge 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 would foster the teams, e.g., by providing frequent interfaces for strategic input.
- *Team—Distributed teams* (Q08): dependencies with not co-located teams might hinder the speed of the Agile one. This includes outsourced components but also teams distributed in different buildings. Especially, when the quantity and the channels of interaction are constrained by contract terms such as time intervals or approval requirement, strong delays occur hindering speed. The same, but less drastic, problems occur by mismatches in processes, practices, and 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, creates silos that hinder the reuse across the projects. The exclusive focus on one project by the team leads also to hindering the interaction (and speed) among teams in different projects: this creates inefficiencies, especially if there are dependencies among the projects, and strategic reuse always brings such dependencies. Solutions such as relocation in different projects have been proposed in the

qualitative answers, to spread knowledge of the system and to gain acquaintance among the employees.

- *Team—Sales unit*: challenge 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 of the similar ones. This suggests that full reusability of components should be checked by the sales unit. This would encompass interaction with the involved teams responsible for the components. The strong recognition of this challenge 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 not optimal decisions.

5.4 Root Factors for Interaction Speed Hindering Business Goals

The previous challenges can be gathered under the name of *interaction speed challenges*. As outlined in the first section, we found that such challenges were impacting different business goals related to speed [6] (Fig. 5.4).

By managing the factors influencing interaction speed, we indirectly aimed at facilitating the achievement of three business goals connected with speed (explained in the beginning of this chapter) and influencing, in the end, return on investments. First we explain how interaction speed influences other kinds of speed (and related business goals):

- *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*.

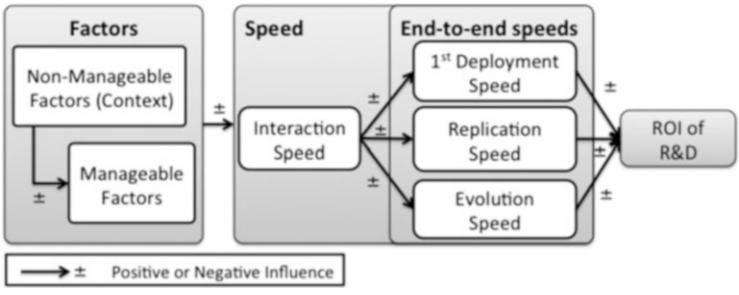


Fig. 5.4 Three kinds of end-to-end speed (influencing return on investments) and their dependency on interaction speed

- *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.

We conducted a root factor analysis and we have found root factors causing different challenges (symptoms). In addition, we identified the information content that was involved in the challenges. Finally, we also captured the influence of ASD. Our findings are described in Table 5.2.

Conclusion

Responsiveness and speed are major challenges for large software industries. These companies are challenged by the demand to minimize the time between the identification of a customer need and the delivery of a solution. ASD is broadly viewed as a solution to this challenge, but we questioned whether the focus on short-term deliveries negatively affects responsiveness in the long run.

In this chapter, we identify three types of speed in development:

- *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 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 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 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.

In this chapter, we studied the root factors causing different challenges (symptoms), which information content was involved, and what was the influence of ASD. By managing the factors influencing interaction speed, we indirectly aimed at facilitating the achievement of the business goals

(continued)

Table 5.2 List of root factors for interaction challenges

F1. Lack of knowledge availability	**Symptoms**: 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. They may have to wait for the expert to be available, and if the expert is overloaded, this might create waste of time (waiting). The team may alternatively decide to make assumptions on the answers that lead to redoing most of the work **Content involved**: The expertise may encompass different kinds of knowledge: domain knowledge is especially requested in embedded systems where software is specific for the device; product architecture, technical knowledge, tool, and process knowledge are also quite requested **ASD influence**: 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 interactions might be
F2. Expert's reputation	**Symptoms**: If an employee has a high reputation of having a specific knowledge, the person will be contacted often. This causes frequent interactions and requests for intervention: the expert is therefore forced to reduce his/her productivity to give support to other teams, and tasks switching might worsen it even more. Loss in expert productivity can be considered a waste, since the value added by an expert to the product is often high and produced quickly. An important point to make is that reputation is not only based on the real knowledge of an employee but rather on his or her social reputation. Consequently the social and informal aspect is what differentiates this factor from F1, where availability depends on the allocation of time formally decided for the expert **Content involved**: The same expertise described in F1 **ASD influence**: ASD principles value social interactions over formal knowledge, amplifying the effects of this factor on interaction speed (as also hypothesized in). 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 in the beginning of development	**Symptoms**: 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 seems to determine the quality of the specification, which influences the elaboration time by the team **Content involved**: Requirement specification and requirement agreement, input for the final value to deliver to the customer

(continued)

Table 5.2 (continued)

	ASD influence: This factor might be connected to the declared support, in ASD, to deal with volatile requirements, i.e., the ones that change often. However, *reacting* to changing requirements and *starting the development on non-clear requirement* are not the same thing and should be clear when employing ASD
F4. Unexpected feature dependencies	**Symptoms**: 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-technical reasons. The team needs to negotiate APIs or to frequently merge changes on a shared component **Content involved**: Change management, change agreement, task distribution **ASD influence**: The dependencies problem is not covered by any known Agile practice. The only mitigation practice already present in Scrum is the presence of a Scrum master for the facilitation of the communication with other teams (through other Scrum masters)
F5. Lack of understanding (*stakeholders' needs*)	**Symptoms**: Large organizations are forced to spread teams in space and co-location is not often possible. The distance and lack of communication between employees with different knowledge and tasks hinder the awareness of each other, which consequently decreases the understanding about each other's needs and alters expectations. This usually causes corrupted communication and corrupted value, and long waiting time (for not recognized urgent tasks) **Content involved**: Communication of needs **ASD influence**: The trend of creating small, self-managed teams strengthens the group development among few individuals, but risks to isolate them from other parts of the organization
F6. Lack of common time	**Symptoms**: 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 **Content involved**: Synchronization, several other kinds **ASD influence**: As for F5, the team might tend to focus on their work only (isolation), disregarding intergroup interaction and not allocating time for such activity
F7. Mismatch of team's styles of communication	**Symptoms**: 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, while the other doesn't reply often to e-mails and is used to communicate through face-to-face meetings.

(continued)

Table 5.2 (continued)

	The effect is a lack of communication. Another issue may be the different uses of knowledge containers such as boundary objects (e.g., wikis) **Content involved**: Intergroup generic communication **ASD influence**: The Agile culture of letting teams have their customized processes somehow encourages this mismatch
F8. Slow resource indexing (*lack of knowledge accessibility*)	**Symptoms**: 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, Fig. 5.1) may be long and therefore delaying **Content involved**: Knowledge about the needed information **ASD influence**: 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 (*commitment*)	**Symptoms**: Once an interaction is needed, the involved parts (single employees or whole teams) have to prioritize the interaction as an ongoing task. If the interaction is considered as "low priority," the team will delay tasks and communication, hindering the other team (s) involved **Content involved**: All **ASD influence**: Isolation of the teams, as explained in F4–5
F10. Interpersonal conflicts	**Symptoms**: 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 **Content involved**: All **ASD influence**: A work environment strongly built on social interactions (as ASD suggests) may amplify this factor
F11. Lack of understanding (*system architecture*)	**Symptoms**: The understanding of the overall architecture might be misinterpreted or miscommunicated by/to the developers. The architecture works as a coordination mechanism, and it's particularly important for connecting hardware and software in embedded systems. The misunderstanding of architecture might cause a lack of architecture conformance, which may lead to unpredictable events causing frequent interactions (see F4, unexpected feature dependencies) during development and maintenance **Content involved**: Architecture knowledge **ASD influence**: The lack of support and explicit practice for architecture focus and management in ASD might lead the team to privilege quick solutions to

(continued)

Table 5.2 (continued)

	conformance, which might speed up the development locally but not globally (e.g., for the whole project/ product)
F12. Lack of understanding (*design*, *technical*)	**Symptoms**: Some roles in the organization, which are not in continuous contact with software designers and programmers, might lack the understanding of specific implementation (e.g., managers or system responsible/ architects). Decisions made ignoring such knowledge might be the prioritization of features or the choices of architectural mechanisms. Such decision creates extra work and extra interactions for the development team (against the ASD principle of optimizing the output) or they might even prove unfeasible **Content involved**: Design and technical knowledge **ASD influence**: The self-management of teams and the trend to avoid documentation might lead to disregard technical and design documentation (at a suitable level of abstraction) influencing other social groups
F13. Lack of awareness (*wrong expectations*)	**Symptoms**: Large organizations are forced to spread teams in space, and co-location is not often possible. The distance and lack of communication between employees with different knowledge and tasks hinder the awareness of each other, which consequently alters expectations **Content involved**: Expectations about the capability of stakeholders **ASD influence**: Isolation, again, is an amplifier
F14. Lack of personal acquaintance	**Symptoms**: The lack of opportunities for co-location decreases the acquaintance existing between two persons in the social network. Such lack creates delayed communication in many situations **Content involved**: All **ASD influence**: ASD is focused in creating and reinforcing such personal acquaintance within the teams but lacks practices for intergroup interactions
F15. Mismatch of processes	**Symptoms**: Embedded software development depends on different processes connected to different parallel engineering practices not software related. As an example, in hardware design usually a V model is used, while stage gates are usually set by product development. Agile teams usually follow short iterations and delivery of software. If the activities involved within the iterations rely on interactions with other groups and other processes, this might cause communication challenges, for example, waiting for feedback dependent on tasks that are not contemplated at that time in other processes **Content involved**: All **ASD influence**: ASD's short iterations are not widespread in other processes followed for other engineering activities and in product management, which cause the proliferation of this factor

connected with speed (explained in the beginning of this chapter) and influencing, in the end, return on investments.

In future work we aim to expand our research to other companies beyond the ones studied in this chapter in order to validate the results. In addition, we seek to develop a framework for companies to proactively design their organizational setup for optimal speed.

References

1. Dingsøyr, T., Nerur, S., Balijepally, V., Moe, N.B.: A decade of agile methodologies: towards explaining agile software development. J. Syst. Softw. **85**, 1213–1221 (2012). doi:10.1016/j.jss.2012.02.033
2. Martini, A., Pareto, L., Bosch, J.: Enablers and inhibitors for speed with reuse. In: Proceedings of the 16th International Software Product Line Conference – Volume 1, SPLC'12, pp. 116–125. ACM, New York. doi:10.1145/2362536.2362554
3. Bosch, J., Bosch-Sijtsema, P.: From integration to composition: on the impact of software product lines, global development and ecosystems. J. Syst. Softw. **83**, 67–76 (2010). doi:10.1016/j.jss.2009.06.051
4. Pikkarainen, M., Haikara, J., Salo, O., Abrahamsson, P., Still, J.: The impact of agile practices on communication in software development. Empir. Softw. Eng. **13**, 303–337 (2008). doi:10.1007/s10664-008-9065-9
5. Martini, A., Pareto, L., Bosch, J.: Communication factors for speed and reuse in large-scale agile software development. In: Proceedings of the 17th International Software Product Line Conference, SPLC'13, pp. 42–51. ACM, New York. doi:10.1145/2491627.2491642
6. Martini, A., Pareto, L., Bosch, J.: Improving businesses success by managing interactions among agile teams in large organizations. In: Herzwurm, G., Margaria, T. (eds.) Software business. From physical products to software services and solutions, lecture notes in business information processing, pp. 60–72. Springer, Berlin (2013)

Chapter 6
A Framework for Speeding Up Interactions Between Agile Teams and Other Parts of the Organization

Antonio Martini, Lars Pareto, and Jan Bosch

Abstract A known problem in large software companies is to balance the return on investment coming from short-term and long-term business goals dependent on the responsiveness of the companies. In the previous chapter we have found challenges in interactions between the Agile team and other parts of the organization. We have conducted an investigation on three large product line companies employing Agile software development in order to find practices that would mitigate the challenges.

6.1 Background

In the previous chapter we have shown the problem of interaction challenges between the Agile teams and other teams or other parts of the organization [1, 2]. Such challenges decrease speed of the companies, hindering the achievement of business goals [3]. In this study we have defined a framework for the mitigation of such interaction challenges. We have asked, through a survey, 36 practitioners about how they (would) mitigate each challenge.

First, it's important to explain the main idea (Fig. 6.1) behind the application of the practices: the development team exchange information content (e.g., clarification of requirements or architecture patterns) via two-way communication with other social groups in the organization (they might be other teams or other roles, like product managers or system architects). The information exchange might be hindered by one or more challenges (exhaustively described in the previous chapter). Such challenges are caused by root factors: by applying practices that change these factors, we aim at improving the information flow. However, we have found during our investigation that the application of the practices might incur in obstacles, which we call barriers.

A. Martini (✉) • L. Pareto • J. Bosch
Chalmers University of Technology, Gothenburg University, Gothenburg, Sweden
e-mail: antonio.martini@chalmers.se; Jan@JanBosch.com

J. Bosch (ed.), *Continuous Software Engineering*,
DOI 10.1007/978-3-319-11283-1_6

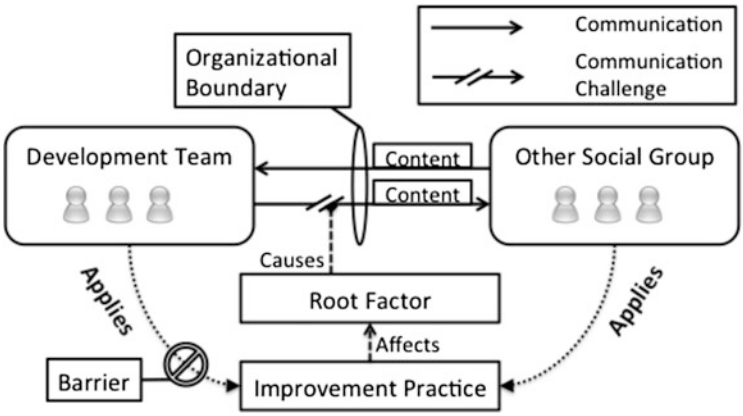


Fig. 6.1 Improvement framework for communication challenges

6.2 Improvement Framework for Mitigating Interaction Challenges and a Use-Case Scenario

The main purpose of our framework is to support a combined effort of managers, teams, and other social groups in detecting visible symptoms, investigating root factors, and applying practices with the aim of improving the achievement of business goals based on speed, as outlined in Fig. 6.2. We propose what might be a typical scenario, where managers want to reach the business goals dependent on speed (first deployment speed, replication speed, and evolution speed described in the previous section) and practices need to be put in place.

In such scenario, delays over the schedules are due to speed wasted in interactions. Managers may recognize it but they need the team(s) and/or the other social group to observe the visible effects (communication challenges). Since each effect is related to one or more root factors, managers can immediately investigate the status of such factors in the teams to find which one is the cause for the effect. The catalog of factors and their connections to visible effects are intended to 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 improvement practices. This overall process is outlined in Fig. 6.2.

We also provide a table (Table 6.1, in the end of this chapter) for the consultation of which practices should be used by which groups in order to mitigate which root factors. This way, the reader can quickly select the suited practices for a given problem. In such table, the "X" means that the practice has been explicitly mentioned by the respondents, while the "+" represent the authors' interpretation that the practice would be useful also for the given factor and roles.

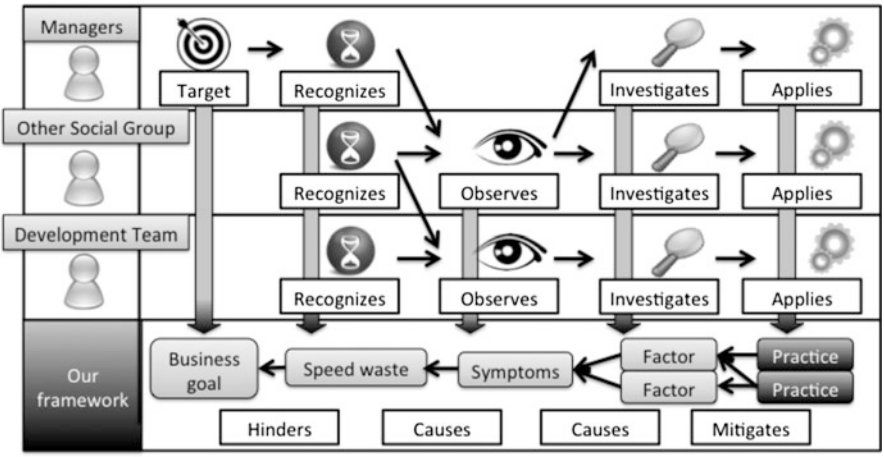


Fig. 6.2 Framework for the management of interaction speed

6.3 Improvement Practices for Mitigating Interaction Challenges

We present a catalog of practices: they don't consist of a complete process description, but rather represent activities to be integrated in (or removed from) an existing process. Our main assumption is that the existing development process is transitioning or an already established version of ASD that needs to be complemented with our practices. For each practice, we list:

Description: which critical boundaries (standing between the team and another social group) are involved, the main activities of the practice, and the connection to the process.

Benefits: a number of positive effects that the practices bring what kind of content is communicated between the social groups and the main underlying negative factors that the practice was supposed to mitigate.

Barriers: the application of the practice might be hindered by some factors. In some cases we omit barriers: this means that we didn't find any clear obstacle in our data.

6.3.1 Practice 1: Integrate Workshops and Meetings at the Start of the Project with the Milestones Defined by the Processes

Members of different teams, with different competences and with different roles, should meet in the beginning of the development for a new software project. The

Table 6.1 Practices address some of the factors and involve specific social groups

		Practices															
1. Meetings at the start of the project	*2. Architecture and requirements in cross-functional teams*	*3. CFT for estimation during product sale phase*	*4. Meetings with architects, system responsible, and testers*	*5. Meetings with project management*	*6. Isolation by programming common available time*	*7. Tailor the processes (with OSSP) for similarity*	*8. Create communication proxies for mismatching processes*	*9. Monitor and define responsibilities for integrated parts*	*10. Include inter-team documentation into the backlog stories*	*11. Create indexes and brokers*	*12. Navigable documentation*	*13. Define intergroup documentation requirements*	*14. Part-time experts serving different teams*	*15. Tools for awareness*	*16. Archi-tectural rules for ASD*		
Social group involved	System engineers/ architects	X	X		X		X	+	X	X		X	X	X	X	X	X
	Product management							X	X				+		X	X	
	Project management	+				X		X	X				+	X	X	X	
	Business unit	+		X				+	+						+	+	
	Distributed team	X					X	X	X	X	X	X	X	X		X	X
	Supplier	+					+	X	X	X	X	X	X	X			X
	Test unit	X			X		X	+	X	X	+	X	X	X	X	X	
	New employee	X								X	+	X	X	+	X	X	
Root factors addressed	Slow resource indexing (lack of knowledge accessibility)	X	X		X			X	X	X	X	X	X	X	X	X	
	Lack of knowl-edge availability		X	+	X				X	+	X	+		+	X	+	
	Expert's reputation	+					+					X			X	X	
	Lack of under-standing (stake-holders' needs)	X	X	X	X	+			X	X	X		X	X		X	+
	Lack of under-standing (sys-tem architecture)	+	X		X					X			X	X	X		X
	Lack of under-standing (design, technical)		X		X	+				X	X		X	X	X		

(continued)

Table 6.1 (continued)

		Practices															
1. Meetings at the start of the project	*2. Architecture and requirements in cross-functional teams*	*3. CFT for estimation during product sale phase*	*4. Meetings with architects, system responsible, and testers*	*5. Meetings with project management*	*6. Isolation by programming common available time*	*7. Tailor the processes (with OSSP) for similarity*	*8. Create communication proxies for mismatching processes*	*9. Monitor and define responsibilities for integrated parts*	*10. Include inter-team documentation into the backlog stories*	*11. Create indexes and brokers*	*12. Navigable documentation*	*13. Define intergroup documentation requirements*	*14. Part-time experts serving different teams*	*15. Tools for awareness*	*16. Architectural rules for ASD*		
	Lack of awareness (wrong expectations)	X	X	X	X	X		+	+					+		X	
	Lack of personal acquaintance	X	X	+	X	X			+							+	
	Mismatch of processes		+	+			X	X	X						+	+	+
	Low prioritized interaction (commitment)	X								X	X				X	X	
	Lack of common time	X		X	X	X	X								X	X	
	Mismatch of team's styles of communication	+				+							+	X	+		
	Interpersonal conflicts	+															
	Unclear requirements in the beginning of development		X		X										X		+
Unexpected feature dependencies		X		X					X	X			+	+		X	

practice consists of organizing workshops and meetings between the team and the system engineers (architects), other development teams (distributed, suppliers), and with the sales team discussing the expectations of each other for the project. The creation of such opportunities has to be integrated into the defined process: milestones and deliveries have to be adjusted with the definition of time allocated for several kinds of interaction. As suggested directly by one respondent: "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."

Benefits: the practice has the following benefits:

- The programmed events will *avoid delays in the integration phase*.
- All the *following communications are facilitated*.
- *Increases organizational awareness*. Helps in adjusting the expectations of the team and its stakeholder: everyone will have information about who are the other stakeholders involved, their capabilities, and needs.
- *Increases awareness of the product*. Increases the information about business goals, architecture guidelines, and an overall picture about the product.

Barriers: this practice needs an upfront investment in terms of time. Since there is no clear evidence that the time spent in the beginning would be gained at the end of the project, it's difficult to convince managers for resource allocation. However, short workshops require less time than re-doing work and re-aligning in the integration phase at the end.

6.3.2 Practice 2: Include Employees with Knowledge of the (Sub-)System Architecture and Requirements in Cross-Functional Teams

A cross-functional team (CFT) consists of a small collection of individuals from diverse functional specializations within the organization. These types of teams may work together for a limited time and their members may also be members of other teams. This is a generic well-known practice that brings many advantages: CFTs are usually employed for feature or component development, in which the team is responsible from the beginning to the end of the development. Therefore, the team needs enough knowledge to elaborate both requirements and quality attributes of the architecture.

Benefits: placing people with this knowledge in the team is the best solution for having the following benefits:

- *More precise estimation of work*. The daily contact with architects, testers, and system responsible makes different roles aware of each other's needs. This avoids superfluous work. Developers and architects become also aware of the effects of their work on each other's work.

- *Chain tool synchronization*. Different tools that are used for different tasks tend to be aligned. This avoids time spent on integrate artifacts from different tools.

6.3.3 Practice 3: Implement Dedicated Temporary CFT for Estimation During Product Sale Phase

ASD usually includes the customer on site and the continuous feedback. However, in large products (especially if delivered on the market and not to specific customers), this is not often the case. Production decisions have to be made and contracts have to be negotiated in advance. In such cases, an option mentioned by many respondents is a temporary CFT for estimation: a team of main stakeholders, i.e., experienced personnel from different disciplines (representatives from architects, developers, testers) should be involved in the discussions about business goals, contract negotiation with the customers, and market scoping.

Benefits: the consultation of such "task force" would:

- Inform the business unit about the estimated *feasibility of the business goals*.
- *Avoid wrong assumptions* in the scope analysis.
- *Increase information about business strategies* in the development team.

Barriers: the major barrier is the willingness of the specialists from different disciplines to accept to sit together with the others. This denotes the presence of "classes" between the disciplines. CFTs could help in mitigating such barrier, but on the other hand, there is the risk of obtaining non-efficient teams. A possible barrier is the feasibility of the integration of this practice in the contract negotiation process or scoping analysis.

6.3.4 Practice 4: Formal and Informal Meetings Between the Development Team and Architects, System Responsible, and Separated Testers

Co-location of system engineers, architects, and testers is not always possible. In such case, meetings are a good practice suggested in many Agile software development approaches such as Scrum. However, often meetings are meant to be within the same team or among the Scrum masters of various teams (e.g., Scrum of Scrums). This practice suggests the participation of system engineers, architects, and testers belonging to a separate unit to formal Scrum meetings in order to reach agreement on requirements. However, formal meetings are not always enough. More than one respondent suggests the actual creation of a social link between the system engineers and the development team.

Benefits:

- *Avoid the mismatch of overall* (*reference*) *architecture and Agile deliveries*: in large projects, architecture is important for reuse, replication, and evolution of the product. Clearly, there is a need to avoid a mismatch between the Agile team's deliveries and the overall architecture.
- *Facilitate the integration of roles with broad responsibilities with ASD*. Roles such as architects and system responsible are not quite emphasized in Agile practices. However, such roles are important to manage complex and large projects.
- *Decrease the written documentation*. Agile suggests the minimization of the documentation: the respondents strongly recommended meetings instead.

Barriers: the main barriers to the implementation of this practice are, as mentioned for the previous practice, political boundaries related to the disciplines.

6.3.5 Practice 5: Formal and Informal Meetings with Project Management

The management team needs to know the delivery progress, the performance of the team, and what decisions have been taken and why. ASD prescribes informal communication and self-organization of the team. One of the informants says: "We have a lot of documents that we produce to keep the management (project, product and line) happy. We then have a completely different set of documents that we produce to remember what we were doing with the code. We usually spend a considerable amount of energy to try to avoid writing the first kind of document and we generally have to hide the writing of the latter documents." The communication should be carried out face-to-face rather than by the delivery of extensive documentation. Management and teams need to define bidirectional communication through meetings in their process, instead of the one-way one provided by the documentation.

Benefits: this practice has the following effects:

- *Decreases the amount documentation for the Agile team*
- *Boosts self-directed teams*
- *Increases the personal acquaintance between the managers and the teams*
- *Increases the strategic awareness of the teams*
- *Increases the awareness of the real current status of the team by the managers*

Barriers: a major barrier to the application of this practice is the mismatch of competences and knowledge between managers and team members: for a manager it might be difficult to catch technical details, which could affect the strategic decisions.

6.3.6 Practice 6: Provide Isolation by Programming Common Available Time (Workshops) with the Critical Groups

Teams need to focus. Despite the importance of face-to-face communication, periods of isolations are strongly claimed as important for the development by the respondents. Therefore, face-to-face communication has to be regulated by programming a regular period of available time common to the groups that are supposed to meet (according to the critical boundaries).

Benefits:

- *Allow teams to focus in the remaining time.*
- *Provide opportunities for meeting with target groups.*
- *Facilitate the management of face-to-face interactions.*

Barriers: the main barrier of this practice is mismatch of calendars and the usual focus of the teams and the other groups on the business goal at hand. Pressure might in fact cause the opposite effect, i.e., the complete isolation of the team, which would be an anti-ASD effect.

6.3.7 Practice 7: Tailor the Processes to Achieve Similarity

Literature and the respondents suggest to tailor the interfaces between the development team and groups with the mismatching processes. "Review interfaces between different processes e.g. System design – SW Design to enable a more continuous flow of information." This practice suggests the involved parties to explicitly meet and adapt their processes in order to make them more similar to each other (not necessarily the same). This could be done, for example, by a formal meeting with the purpose of defining common milestones or of creating communication proxies (see next practice).

Benefits:

- *Increase opportunities for communication*. There are usually at least two levels in the hierarchy of the processes, organization (focused on the development of the product with embedded software), and projects. With ASD the teams are meant to be self-organized and following their own processes. Also, processes between different development teams may mismatch. In all these cases, the mismatch of the processes creates lack of communication opportunities: the more similar the processes, the better synchronization and alignment and therefore better chances of communication (Fig. 6.3).
- *Avoid overregulation of team processes*. The Agile manifesto suggests "people over processes": in large organization, where some high-level processes need to

be in place, the support for ASD could be given by the adaptation of the processes to particular needs.

Barriers: well-established and complex processes are already running in a large organization, and many products depend on them. The tailoring of OSSP, even if gradual, represents a big effort involving a large number of employees, which brings costs in terms of time and resources. Different mind-set and political dynamics would also hinder the application of this practice.

6.3.8 Practice 8: Create Communication "Proxies" Between Mismatching Processes

In the previous practice we propose to tailor the processes. Another (complementing) idea is to provide a "proxy process" responsible for the communication between the mismatching processes (Fig. 6.4). This can be achieved by having an Agile facilitator responsible for handling the communication with the Agile team in the other groups or by defining generic regular interactions between the two groups. This concept is already suggested in GSD literature, where a member of a team (Scrum master or team leader) is responsible to communicate to the same role in the other team. However, a requirement in this practice is that the Agile facilitator should know ASD.

6.3.9 Benefits:

- *Separation of tasks synchronization and communication needs*. In different processes with different phases, needs for communication can be satisfied, at least partially.
- *Faster feedback for the Agile teams*. They don't have to wait according to the different processes.
- The Agile facilitator or the meetings contribute to *spread process knowledge* (*ASD*) to the other parties involved.

Barriers: the main challenge is to find a person in the non-Agile group that is aware of ASD and can "translate" the needs of Agile to the rest of the group. The person also needs to understand both parties' needs. In some cases, such role would need to be trained, which would take time. Another barrier is represented by the mind-set of the groups and by diverging "political" dynamics.

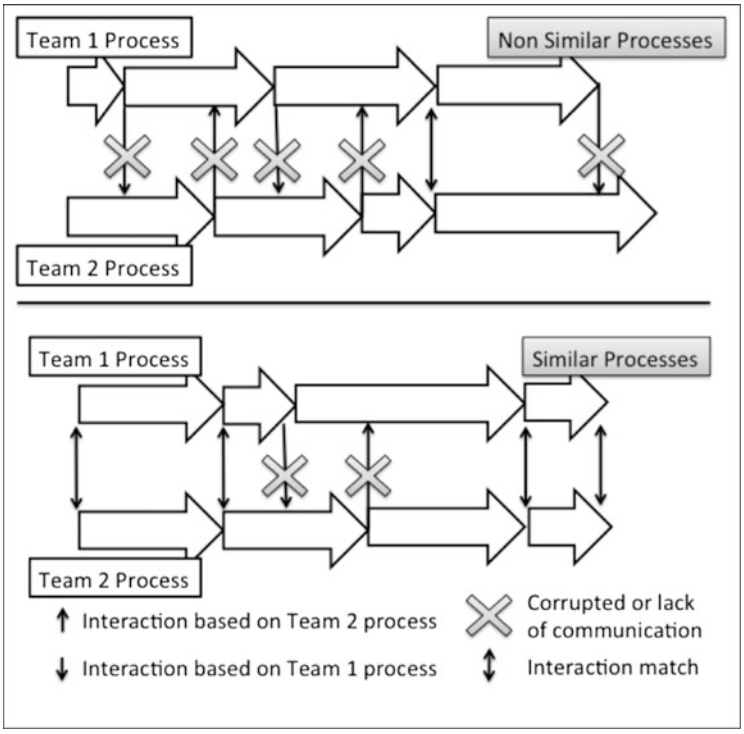


Fig. 6.3 Different flows of communication between different team's processes

6.3.10 Practice 9: Monitor and Meet to Define Responsibilities for Integrated Parts of the Software

Citing the respondents: "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." The respondents suggest that responsibility for integrated parts of the software has to be recognized by the involved parties. The integrated "whole" might be a component (in case of features spread to different components) or a set of connected functionalities. The practice consists in monitoring and recognizing such situations in which responsibility could "fall between the chairs" and reacting by defining responsibilities through a meeting with the involved actors.

Benefits:

- *A clear reference for bug fixes and improvement needs*
- *A clear reference for explanations of decisions*, especially in lack of other kinds of documentation

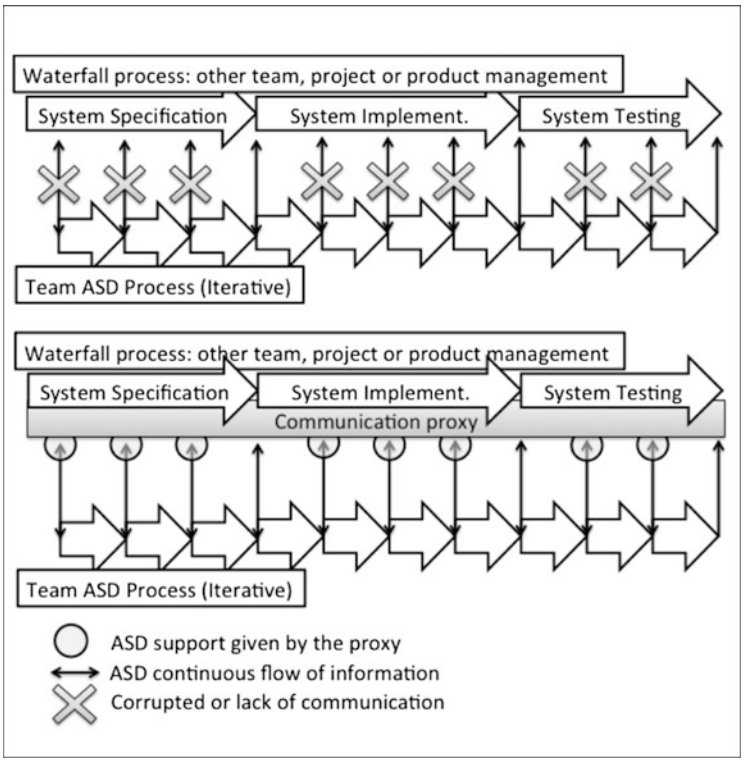


Fig. 6.4 Processes need interfaces for ASD continuous communication

Barriers: one challenge for the implementation of this practice is the tendency in Agile to disregard formal documents. Moreover, in some cases the code has to be touched by so many teams that it might be too difficult to define who is clearly responsible.

6.3.11 Practice 10: Include Inter-team Documentation (Integration-Related) Into the Backlog Stories

"Bring in documentation stories into the backlog. Include documentation into definition of done." The backlog should contain stories of documentation describing the interaction of integrated parts.

Benefits:

- The *prioritization of the documentation production*
- The *increment of technical documentation* provided by the team for the *external stakeholders*

Barriers: the barriers are related to the responsibilities and commitment of such practice: who should be responsible for updating the backlog? The main hypothesis would be the team leader or Scrum master, after the collection of feedback from other teams. Another hypothesis would be an architect, aware of the possible challenges due to integration.

6.3.12 Practice 11: Create Indexes and Brokers

The practice proposed by the respondents is to keep an index of key people with the role of brokers. Such brokers should, in turn, be able to identify the most knowledgeable person for a requested information. "The biggest problem is that it is usually impossible to find the documentation. It should be easily accessible and there should be people ('librarians') who can help you find relevant information or know who are the experts to ask."

Benefits:

- *Avoids unnecessary communication and wasted time* in trying to retrieve knowledge from the wrong persons
- *Decrease the amount of documentation* to be maintained for the Agile team

Barriers: the major threats for the implementation of this practice are the need for the explicit allocation of employees for the given role and the loss of knowledge due to such employees leaving the organization.

6.3.13 Practice 12: Navigable Documentation

Whenever documentation is motivated, it should be made navigable. Many respondents suggest structured documentation: different levels of details for different levels of understanding: "have less complimentary documentation, in best case only some pictures to explain the overall idea." This is referred to code: "All diagrams that directly describes code should be generated from the code as they only serve as another view of the code, they should also be interactive to enable developers to easily switch between a diagram based view of the code and the actual code." The same principle holds for requirements and for context knowledge.

Benefits:

- The structure should provide *a quicker access to requirements*, *code*, *and context knowledge*.

Barriers: the major barrier for this practice is the introduction of a suitable tool that should be integrated with the rest of the infrastructure without creating conflicts.

6.3.14 Practice 13: Define Documentation Requirements for Groups Across Organizational Boundaries

This practice suggests the creation of a set of requirements expressed in terms of goals. Such requirements should be provided for each frequent consumer of a kind of documentation: for example, technical documentation shouldn't be provided for managers, but it would be useful for checking architecture compliance and avoid erosion: therefore, the architects should define requirements for the documentation produced by the team for them. The definition, according to the Agile principles, should take place through a meeting between the involved parties to discuss the requirements.

Benefits:

- *Defines clear constraints for the documentation*, avoiding the explanation of not relevant details.
- The producer of the documentation *gains awareness of the needs of the stakeholders*.

Barriers: the major barrier for this practice is that such documentation requirements are not well known and there is a need for guidelines to define documentation requirements, but further research has to be carried out to provide them with respect to different interfaces.

6.3.15 Practice 14: 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/her an inner consultant serving the other teams. This involves a process of identifying the critical knowledge, allocating time to the expert broadcasting the information of such availability to the teams.

Benefits:

- Grouping interactions in a defined time box would *avoid high frequency of interactions*.
- Provides every Agile team with a *clear and dedicated resource of knowledge*.

6.3.16 Practice 15: Increase the Tools for Awareness

Agile teams would benefit from the visualization of social network flows, knowledge location and availability, and time availability.

Benefits:

- Knowledge visualization would increase the awareness of the environment surrounding the involved Agile teams, *avoiding isolation*.
- Shared calendars would provide a good way to check the availability of the employees with the knowledge and therefore *increases the opportunities for creating interaction* when needed.

Barriers: such visualization tools have to be integrated with current tools, e.g., mailing services, which might not have such features. Such integration might not be available or easy to apply and maintain.

6.3.17 Practice 16: Implement Architectural Rules for ASD

Agile teams developing different components (or features) need to be decoupled as much as possible. This suggests the need of architectural design rules to match the organizational (Agile) structure with the architectural components. With the introduction of feature-oriented teams, the component-oriented architecture might cause loss of responsibility for components, causing loss of architectural care. For component teams, the spread of features across different teams would increase the number of interactions among the teams for requirements and design agreement.

Benefits: the informants suggest the need for new architectural solutions to *combine* the following goals:

- Allow the teams to *focus on the customer value*.
- Avoid *organizational dependencies* caused by the architectural ones (loose coupled teams).
- Satisfy system requirements but adapt and take into account information about software design.

Barriers: the implementation of a suitable architecture compatible with ASD remains an open issue in research and industry.

Conclusion

In this chapter we presented a framework for the mitigation of interaction challenges between Agile teams and other teams or other parts of the organization. As a research method we employed a survey where we received responses from 36 practitioners about how they (would) mitigate each type of identified interaction challenge.

The main purpose of our framework is to support a combined effort of managers, teams, and other social groups in detecting visible symptoms, investigating root factors and applying practices with the aim of improving

(continued)

the achievement of business goals based on speed. We propose what might be a typical scenario, where managers want to reach the business goals dependent on speed (first deployment speed, replication speed, and evolution speed described in the previous chapter) and practices need to be put in place.

In such scenario, delays over the schedules are due to speed wasted in interactions. Managers may recognize it but they need the team(s) and/or the other social group to observe the visible effects (communication challenges). Since each effect is related to one or more root factors, managers can immediately investigate the status of such factors in the teams to find which one is the cause for the effect. The catalog of factors and their connections to visible effects are intended to 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 improvement practices. We also provide a table (Table 6.1, in the end of this chapter) for the selection of the practices that should be used by each group in response to identified root factors.

In future work, we aim to broaden the scope of the study to include more companies and respondents as well as explore if there are additional practices to be considered.

References

1. Martini, A., Pareto, L., Bosch, J.: Enablers and inhibitors for speed with reuse. In: Proceedings of the 16th International Software Product Line Conference – Volume 1, SPLC'12, pp. 116–125. ACM, New York (2012). doi:10.1145/2362536.2362554
2. Martini, A., Pareto, L., Bosch, J.: Communication factors for speed and reuse in large-scale agile software development. In: Proceedings of the 17th International Software Product Line Conference, SPLC'13, pp. 42–51. ACM, New York (2013). doi:10.1145/2491627.2491642
3. Martini, A., Pareto, L., Bosch, J.: Improving businesses success by managing interactions among Agile teams in large organizations. In: Herzwurm, G., Margaria, T. (eds.) Software Business. From Physical Products to Software Services and Solutions, Lecture Notes in Business Information Processing, pp. 60–72. Springer, Berlin (2013)

Chapter 7
Customer-Specific Teams for Agile Evolution of Large-Scale Embedded Systems

Helena Holmström Olsson, Anna B. Sandberg, and Jan Bosch

Abstract Companies serving multiple customers or a segmented mass market often struggle with two conflicting forces. On the one hand, companies need to deploy a constant stream of new features to their customer base. On the other hand, individual customers demand rapid feedback to their request for dedicated functionality. Although the conflict of achieving scale and at the same time staying responsive to individual customers has been recognized by others, it is particularly challenging in large-scale software development. In this paper, we study the concept of customer-specific teams (CSTs) as an answer to this challenge. The CST notion builds upon agile values and is an effective means to shorten feedback loops and increase customer responsiveness, customer satisfaction, and feature quality. Also, the approach allows for more innovative feature development bringing with it new business opportunities in a market where competition is fierce. We illustrate and validate the approach in the context of one of the business units of Ericsson.

7.1 Introduction

For more than a decade, agile development methods have proved successful for establishing flexible development processes with short feedback loops and close customer collaboration [1, 2]. Due to successful accounts [3, 4], these methods have become attractive to a broad variety of companies. Currently, large software-intensive

H.H. Olsson (✉)
Department of Computer Science, Malmö University, Malmö, Sweden
e-mail: helena.holmstrom.olsson@mah.se

A.B. Sandberg
Ericsson AB, Gothenburg, Sweden
e-mail: anna.sandberg@ericsson.com

J. Bosch
Department of Computer Science and Engineering, Chalmers University of Technology, Gothenburg, Sweden
e-mail: Jan@JanBosch.com

J. Bosch (ed.), *Continuous Software Engineering*,
DOI 10.1007/978-3-319-11283-1_7

organizations are in the process of deploying agile methods, and a number of attempts to scale agile methods can be identified [5, 6]. However, to apply these methods in large-scale development of products intended for a mass market is not without challenges. In large-scale software development, there is typically a conflict between responsiveness to individual customers and scale in terms of delivering a high number of features to as many customers as possible. Most often, organizations focus on scale, and individual customer requests are viewed as problematic since they add complexity to product variation and version control.

In our study, we focus on how to increase responsiveness to individual customers in large-scale software development without losing focus on scale. We do so by studying CSTs at Ericsson, i.e., development teams that work exclusively with prioritized customers to quickly respond to their particular needs. The intention with these teams is to help the organization to shorten feedback loops and increase responsiveness as advocated by agile methods. At the same time, the teams contribute to scale as soon as the value they produce in customer-specific collaborations is transferred into the main product branch and made available to the larger customer base.

7.2 Large-Scale Software Development

For more than a decade, agile development methods have demonstrated their success in increasing flexibility and reducing development lead time [1, 2]. Due to successful accounts [3, 4], agile methods have become attractive to a broad variety of companies. Currently, large software-intensive organizations are in the process of deploying agile methods, and attempts to scale agile methods are common [5–7]. However, the applicability of these methods is not without challenges in large-scale software development [8]. As recognized by Badampoudi et al. [9], organizations often discover misalignments between methods when attempting to apply agile methods in a large-scale setting. The reason for this is that the translation of the original agile ideas to a large-scale setting is very difficult. Also, the shift towards agile is difficult for companies that are used to heavyweight sequential processes and that have interdependent teams and stakeholders located at different locations [9]. Often, distributed development teams lack a shared understanding due to communication and coordination challenges, lack of documentation, and complex decision-making processes.

Another difficulty is the challenge related to cross-functional team creation [8]. To create generalist teams that can implement features in all software components has shown difficult in large systems with high complexity. Often, organizations realize that many components in a large-scale system are technically very difficult and interdependent and require years of experience to be fully understood by developers. As a result, many large-scale organizations experience long lead times before the development teams can implement anything useful in a

component, and to identify who has the required extensive expertise to perform a task is still a challenge in a large-scale adoption of agile methods.

7.2.1 Scale Versus Responsiveness in Large-Scale Agile

Companies involved in large-scale software development deliver systems to a significant number of customers. Typically, these customers have different ideas on how the product can serve their particular needs. The role of product management is to inventory these needs; to combine, merge, and prioritize among these; and to present a roadmap with a set of requirements for the next release of the system. In this process, strategic customers are looking to include those requirements that are important to them. This leads to a tension between two conflicting interests. On the one hand, the development organization needs to achieve scale in terms of implementing as many new features to as many customers as possible. On the other hand, the development organization needs to show responsiveness to strategic customers and minimize the delay between a customer request and the deployment of a solution that meets this request. Most organizations focus on achieving scale, and customers that ask for unique solutions are viewed as problematic [10].

To better understand the tension between scale and responsiveness, we introduce two concepts, i.e., "customer-unique features" and "customer-first features." A customer-unique feature is a feature that has relevance to one specific customer, and the likelihood of any other customer requesting the same feature is low. A customer-first feature, on the other hand, is a feature requested by one customer, but that has relevance for other customers and therefore can be transferred into generic functionality as part of the product main branch.

7.2.2 Customer-Specific Team Development

Customer-specific teams are cross-functional teams that work closely with prioritized customers. The notion of CSTs is well established in areas such as product sales [11], customer relationship management [12], and customer support [13], and these teams have proven useful for improving long-term relationships and for adding value to customers. Customer-specific teams in software development typically include competencies ranging from architects and system designers to software developers and testers. With end-to-end responsibility for feature development, CSTs allow for an autonomous development organization that rapidly responds to individual customer needs. While these teams do not alter the planned release cycle, they can rapidly develop new features and insert these in the release process, allowing for significant improvement in response time to customer-specific requests. Also, CSTs allow the development organization to tap into valuable

customer knowledge and to learn about the specifics of individual customers. If successfully used, this knowledge can help companies understand not only one customer but also other customers with similar needs.

7.3 Research Site and Method

This study was carried out in close collaboration with Ericsson AB, a world-leading provider of telecommunication systems. Ericsson has been transforming its development organizations towards agile development since 2005, and today agile practices have become the de facto way of working for several of the product development units. Ericsson has cross-functional development teams that are accountable for a feature from the formulation of requirements until release to customers. As a result of this autonomous team structure, a team can work as a customer-specific team for a period of time. From being more of an experiment a few years ago, CSTs have become a natural part of Ericsson's development organization. Today, the development organization consists of roadmap teams producing features for the generic product with scale being the primary metric of success and CSTs operating to increase responsiveness to individual customers. As a result of a successful balance of roadmap teams and CSTs, the development organization has the ability to do (1) roadmap development, i.e., features for the generic product; (2) customer-unique development, i.e., features customized for one customer; or (3) customer-first development, i.e., features requested by one customer but with relevance to the main product branch.

The development organization involved in this study has 30 cross-functional teams, located at two different sites in different time zones. Each cross-functional team consists of 7–8 members. To manage releases, program management, feature integration management, and release management support the teams. All teams are involved in the development of one of the nodes in the 3G networks that includes customer-requested features as well as support for mobility management. For Ericsson, finding a balance between roadmap and CSTs is important. Today, this balance depends on the amount of customer-first requests. Still, development of customer-first features is limited, and roadmap teams are still in majority. From a development perspective, it should be noted that most cross-functional teams prefer development of new roadmap features for the generic product, over development of a new feature on an old configuration for a specific customer. Due to this, most cross-functional team members do not favor customer-specific development more than classical roadmap development although the common view is often that customized development is more attractive due to its short-term focus.

In total, we conducted 17 interviews and one group interview. In the interviews, we met with a team leader, a system manager, a system designer, and a function tester from three CSTs, as well as people at two customer units with which these CSTs interact. Furthermore, we conducted interviews with a program manager, a product manager, and an integration manager at the main development site. In

Table 7.1 Summary of the advantages with having customer-specific teams

	Customer responsiveness	Customer satisfaction	Feature quality
Team A	Short feedback cycles Closer to a specific customer Direct communication	Strong support after delivery Regular meetings with customers Improved interaction with customer units	Continuous discussion about requirements More frequent acceptance tests More frequent usability tests
Team B	Give important customers what they want Bypass the normal release process	Better understanding of feature usage Anticipate problems Increased customer control	Opportunity to test in advance Test in field—learn more about specific customer needs
Team C	Increased flexibility Faster deliveries Adapt faster to customer needs	Customize features Improved understanding of feature usage Customers get what they want when they want it	More adapted and flexible processes and testing procedures
Customer Unit A	Closer to developers Direct communication Short feedback loops	Customers get extra attention	Test directly in the field with a specific customer
Customer Unit B	Faster deliveries Customers decide when to get a feature	Better understanding of feature usage	Learn what requirements mean instead of having assumptions
Product Manager	Improved interaction between developers and customers	Better understanding of customers	Learn about specific customer needs Facilitate good testing
Program Manager	Increased responsiveness Less disruptive	Only value-adding development	Bundle customer-specific features
Integration Manager	Special treatment for important customers Bypass release cycles	Better understanding of customers Develop the right things	Better understanding of quality

addition, and as a follow-up to the interviews, we conducted a group interview in which we met with three managers at the main development site. All the interviewees have significant experience from Ericsson. For the purpose of our study, they provided a rich account on benefits as well as challenges with customer-specific team development, and due to their extensive experience, they were all able to compare today's situation with the time before CSTs. In Table 7.1, we summarize our interview findings.

7.4 Findings

Customer-specific teams increase responsiveness to individual customers by enabling fast development of "customer-first" and "customer-unique" features. This is difficult to achieve in traditional roadmap development where features are released in planned release cycles adapted to generic needs and requests. During the interviews, one of the product managers refers to CSTs as twice as fast as roadmap teams. The reason for this significant improvement of speed is that CSTs don't need to package their features into the regular bi-yearly release cycle in which administration and coordination aspects are time-consuming.

In relation to customer satisfaction, the opportunity to learn about feature usage is valuable. Our respondents emphasize the opportunity to understand what features that are used and those that are not and why. Furthermore, CSTs enhance customer satisfaction by allowing customized functionality for prioritized customers. While customer satisfaction is as important in traditional roadmap development, it is difficult to evaluate since direct customer contact is scarce.

Finally, CSTs improve feature quality by facilitating continuous feedback and daily communication with customers. Such a dialogue is difficult to achieve in traditional roadmap development and especially in a large-scale setting where frequent dialogue with a large number of customers is difficult to establish and maintain. In Table 7.1, we summarize our interview findings.

Overall, the experiences of CSTs are positive. Our interviewees are familiar with classic roadmap team development where focus is on scale, and in comparison with this, CSTs allow for significantly faster responsiveness to individual customers. In particular, CSTs prove beneficial when a customer wants a regular feature fast and is willing to pay for it. Also, CSTs are used to provide extra support to prioritized customers.

However, our study shows on three main concerns that need to be considered when adopting CSTs. First, there are challenges related to resource allocation. While a customer-specific team can always start working on a roadmap feature if not busy with a customer-specific request, this will be on cost of speed if a customer-specific request occurs. Second, by establishing a close relationship to customers, the development organization exposes itself in ways not experienced in traditional development. As a result, customers might view CSTs as a general support function to which they turn for discussing any problem that might occur. For a customer-specific team, this has a negative impact on development speed for the feature they develop. Third, agile practices such as continuous integration and continuous regressions testing are vital. If these mechanisms are not fully in place, there is cumbersome manual work to do whenever features need to be integrated into the main code base.

7.5 Discussion

7.5.1 Feature Development Approaches

As a result of our study, we identified three approaches in which to organize feature development (see Fig. 7.1):

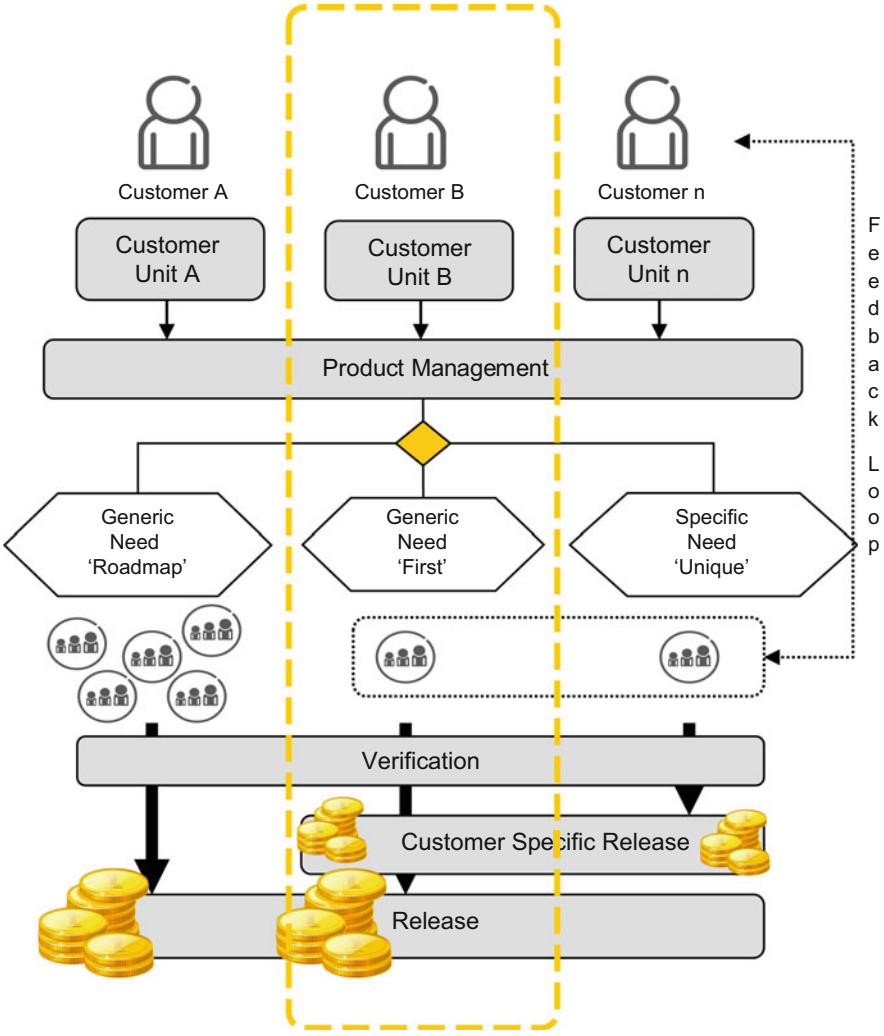


Fig. 7.1 Three approaches for feature development in large-scale Agile development

1. The first approach is classic roadmap development where the focus is on scale. We name this approach "generic need roadmap," characterized by development according to planned release cycles.
2. The second approach is "customer first" and represents the opportunity to have CSTs develop features for individual customers that soon transfer to generic product functionality. This approach has a huge business potential as the company gets paid when the feature is delivered to the customer that requested it, as well as when delivered as generic functionality as part of a roadmap release.
3. The third approach is "customer unique" and reflects a situation in which CSTs develop features unique for one customer, but without relevance to the generic product.

7.5.2 Customer-Specific Team Approaches

To further understand how CSTs complement already established practices and the way in which they can be used to advance the practices of large-scale software development organizations, we suggest focusing on the dimension of *responsiveness* and how this contributes to *innovativeness*. Both responsiveness and innovativeness [1–3] are key attributes for successful team output. Similar with our findings on team responsiveness (see Sect. 7.4), we refer to responsiveness as "the ability to respond to customer requests." Also, and as suggested by our interviewees, CSTs enable the organization anticipate future requests. In this discussion, we refer to this as innovativeness and "the ability to actively learn about new feature needs."

In Fig. 7.2, we illustrate the dimensions of responsiveness and innovativeness and the different approaches to customer-specific team development that an organization can choose: (1) feature-boxed development, (2) opportunity-based development, (3) backlog building development, and (4) continuous innovative development.

Feature-boxed development is when a customer-specific team develops a feature in close collaboration with one specific customer. In this collaboration, the focus is on one feature only, and there is no systematic process in place to respond and act on requests outside the assigned featured. There is also no systematic process in place to actively search for new features. While it is fair to assume that the close collaboration between the development team and the customer has potential to identify opportunities for new feature development, the lack of structured processes to cater for this hinders teams from acting in that way. Although CSTs have detailed knowledge about customers and what could be an opportunity for new feature development, this knowledge must be captured in structured processes so that ideas can be efficiently incorporated into the development and release process. In customer-specific development as referred to here, responsiveness is prioritized over innovativeness, and therefore, teams focus on delivering

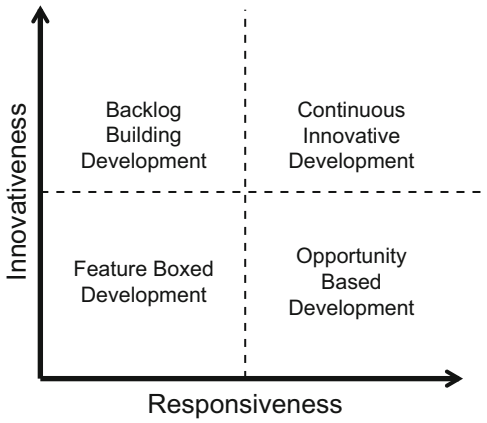


Fig. 7.2 Four variants of customer-specific team development

what has been assigned to them and processes support them in doing this as fast as possible.

Opportunity-based development is when the customer's need for new business opportunities, made possible by new features, is the primary focus. In such collaborations, the customer-specific team is responsive to any request or need that appears during the development process, and there is process support for managing new feature development in place. While the development team does not actively search for what could be new features, they adjust to whatever opportunity that emerges during the collaboration in order to start development of new features as soon as they have completed the original assignment. When choosing this approach, the overall assignment given to the customer-specific team is to focus on supporting the customer's overall business needs over developing a single feature.

Backlog building development has the potential to identify new features. In this approach, a systematic process to actively search for new features is established. Teams work proactively with customers to identify new feature opportunities that can support the customer's business goals. However, while the teams have the ability to identify new features, a systematic process for translating these opportunities into software functionality is still missing. As a result, this approach allows development teams to do "innovative feature-boxed development," but without fast responsiveness due to the lack of processes support. If choosing this approach, organizations use CSTs to generate innovative ideas for new feature development, rather than prioritizing fast development of these.

Continuous innovative development is the "ideal" approach to customer-specific team development in which both responsiveness and innovativeness are present. In such collaboration, the development team actively identifies opportunities for new feature development and responds to these as soon as they have completed development of the previous one. Both the customer-specific team and the customer are open to, and aware of, the processes to actively respond to new

needs and requests. During the collaboration, there is a constant sharing of information between the team and the customer, and this information works as the foundation for short development cycles and idea generation.

To maximize the potential of CSTs to be both responsive and innovative as suggested by the "continuous innovative development" approach, it is important to put systematic process support in place. Such a process needs to include (1) methods for how to identify new features, and (2) an assignment process which allows teams to continue development as new features are identified. With such process support in place, organizations can benefit from efficient development characterized by both responsiveness and innovativeness. For example, when combining continuous innovative development with the concept of "customer-first" feature development, the potential is endless for both the customer and the development organization.

Conclusion
While the conflict of achieving scale and at the same time staying responsive to individual customers is nothing new, it is particularly challenging in large-scale software development. The notion of CSTs builds upon agile values and is an effective means to shorten feedback loops and increase customer responsiveness, customer satisfaction, and feature quality. Also, the approach allows for more innovative feature development bringing with it new business opportunities in a market where competition is fierce.

References

1. Larman, C., Vodde, B.: Scaling Lean and Agile Development: Thinking and Organizational Tools for Large-Scale Scrum. Pearson Education Inc., Boston (2009)
2. Highsmith, J., Cockburn, A.: Agile software development: the business of innovation. Softw. Manag. **34**(9), 120–122 (2001)
3. Abrahamsson, P., Warsta, J., Siponen, M., Ronkainen, J.: New directions on Agile methods: A comparative analysis. In: Proceedings of the 25th International Conference on Software Engineering, pp. 244–254. Springer, Portland (2003)
4. Olsson, H.H., Alahyari, H., Bosch, J.: Climbing the "Stairway to Heaven": A multiple-case study exploring barriers in the transition from agile development towards continuous deployment of software. In: Proceedings of the 38th Euromicro Conference on Software Engineering and Advanced Applications, IEEE, Cesme, 5–7 September 2012
5. Kerievsky, J.: Industrial XP: Making XP work in large organizations. Executive report in Agile Project Management, vol. 6, no. 2
6. McMahon, P.E.: Extending agile methods: A distributed project and organizational improvement perspective. In: Proceedings of the 17th Annual Systems and Software Technology Conference, Salt Lake City, 18–21 April 2005
7. Lagerberg, L., Skude, T., Emanuelsson, P., Sandahl, K., Stahl, D.: The impact of agile principles and practices on large-scale software development projects: a multiple-case study of two projects at Ericsson. In: ESEM, 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 348–356 (2013). doi:10.1109/ESEM.2013.53

8. Heikkilä, V., Paasivaara, M., Lassenius, C., Engblom, C.: Continuous release planning in a large-scale scrum development organization at Ericsson. In: Baumeister, H., Weber, B. (eds.) Agile Processes in Software Engineering and Extreme Programming, vol. 149, pp. 195–209. Springer, Berlin (2013)
9. Badampudi, D., Fricker, S., Moreno, A.: Perspectives on productivity and delays in large-scale Agile projects. In: Baumeister, H., Weber, B. (eds.) Agile Processes in Software Engineering and Extreme Programming, vol. 149, pp. 180–194. Springer, Berlin (2013)
10. Bosch, J.: Maturity and evolution in software product lines: Approaches, artefacts and organization. In: Proceedings of the Second Software Product Line Conference (SPLC2), pp. 257–271, Springer-Verlag London, UK (2002). ISBN:3-540-43985-4
11. Arnett, D.B., Badrinarayanan, V.: Enhancinf customer needs-driven CRM strategies: core selling teams, knowledge management competence and relationship marketing competence. J. Pers. Sell. Sales Manag. **25**(4), 1015–1024 (2005)
12. Chalmeta, R.: Methodology for customer relationship management. J. Syst. Softw. **79**(7), 1015–1024 (2006)
13. Rathnam, S., Mahajan, V., Whinston, A.B.: Facilitating coordination in customer support teams: a framework and its implications for the design of information technology. Manag. Sci. **41**(12), 1900–1921 (1995)

Part III
Continuous Integration

This part discusses the third step on the Stairway to Heaven, i.e., the implementation of continuous integration as a practice for the entire R&D organization. There are four chapters in this part. The first chapter introduces the Continuous Integration Visualization Technique (CIViT) model. CIViT provides a mechanism for visualizing all testing activities in a product R&D organization, ranging from frontline engineers to the release of the product to customers, in one simple graph. This allows teams to communicate about the end-to-end testing activities as well as prioritize the improvements to the continuous integration environment. The second chapter is concerned with the build and integration flows in large software systems where dozens of subsystems need to be integrated into one product. The build systems and the integration flow of part into the final product need to become continuous in a CI context, and the focus of the chapter is this particular challenge. The third chapter is concerned with another challenge in continuous integration: the testing of software in cyber-physical systems where the system contains mechanical and hardware parts in addition to the software. Continuous integration of software is then concerned with using different techniques to accomplish testing without the presence of the physical mechanics and hardware. Finally, the fourth chapter is concerned with an area of testing that has been notoriously hard to automate: visual graphical user interfaces. The chapter discusses industrial experiences and lessons with visual GUI testing as well as the various generations visual GUI testing that have resulted in a third generation that is well applicable in a wide range of industrial applications.

Chapter 8
The CIViT Model in a Nutshell: Visualizing Testing Activities to Support Continuous Integration

Agneta Nilsson, Jan Bosch, and Christian Berger

Abstract Nowadays, innovations in many products ranging from customer electronics to high-end industry electric/electronic components are driven by software. Thus, new or extended features to software and mechatronic products can be realized and deployed to the market much faster. While the use of software enables an enormous flexibility, mastering the ever-growing complexity of the resulting products to meet the quality goals required for the market is getting more and more challenging. Continuous development combined with continuous testing is a successful method that actively incorporates the customer to get feedback for the feature to be deployed early, and thus, product owners, developers, and testers can collaborate more effectively to meet the market's needs. From literature, setting up such an agile development process is clear; the individual situation in terms of organization, processes, and development and test tooling however is depending on the company—many of the aforementioned aspects have grown over the years and cannot be easily changed. In this article, we present the CIViT model, which allows companies to get an explicit understanding and overview of their current testing and integration activities. With CIViT's intuitive representation of the current status, companies are able to identify bottlenecks and derive actions points to evolve their processes, methods, and development and test tooling towards a more agile and continuous deployment-oriented organization. Thus, they will be able to develop, integrate, evaluate, and deploy new features faster to the end user, hence strengthening their own market position.

A. Nilsson (✉) • J. Bosch • C. Berger
Division for Software Engineering, Department of Computer Science and Engineering, Chalmers University of Gothenburg, Gothenburg, Sweden
e-mail: agneta.nilsson@chalmers.se; Jan@JanBosch.com; christian.berger@chalmers.se

J. Bosch (ed.), *Continuous Software Engineering*,
DOI 10.1007/978-3-319-11283-1_8

8.1 Introduction

Software-driven products enable nowadays more flexible product innovations and shorter product development cycles to serve the markets needs. Thus, companies can react on changing customer requirements by extending and evolving the software over time—even in growing product families. While software drives this enormous flexibility for today's companies, mastering the required development and test organization around such a potentially ever-growing product family is an important and urgent challenge for today's companies to avoid the erosion of software, for example.

The development of complex software-driven electric/electronic systems for example is usually broken down into developing software components, integrating them with hardware components to subsystems, before these subsystems comprise the entire product. These different stages are conducted by separate departments within an organization, which in many cases have diverse test methods and test systems to evaluate the quality of their individual contribution.

To shorten product development cycles for reacting faster on the market needs, the overall testing strategy for the entire product needs to be adjusted in such a way that they enable faster feedback—at least twofold: (a) are the evolved product features meeting the customer requirements (acceptance tests)? And (b) when changing and integrating new features starting on the software component level, is the existing functionality preserved or did they introduce unwanted and eventually faulty side effects?

These two questions can be addressed with a thoroughly implemented continuous integration strategy, where developers get fast feedback whether their contributions are working as expected. However, implementing such a continuous integration strategy is challenging—especially for electric/electronic systems where also hardware components are involved in different tests. Furthermore, besides technical aspects, which need to be tackled to realize a fast continuous integration strategy, all "building blocks" of a company's test strategy at the various aforementioned levels need to be coordinated, harmonized, and finally speeded up.

In this article, we are presenting our visualization approach "CIViT model" as a tool to intuitively map the current status of the various test efforts around a specific product and the characteristics of the tests at the various stages. Thus, developers, tests, and managers can easily spot the current quality of a product's test strategy and identify where the test strategy needs to be better coordinated and harmonized to speed up the product development and deployment cycles.

The rest of the article structured as follows: In Sect. 8.2, related work is presented and discussed. In Sect. 8.3, we describe the foundations and goals for our visualization approach "CIViT model." Section 8.4 presents results from applying the CIViT model to one of the companies in the Software Center, while Sect. 8.5 describes its evaluation. Section 8.5 summarizes the article and gives an overview of potential future work.

8.2 Related Work

Speeding up the development and deployment of high-quality products needs a well-adjusted test and integration strategy. The realization of such a strategy is called continuous integrations, which enables developers to realize new features or improve existing ones, while his or her contributions are directly subject to further—and in the best case automated—testing on the various product stages. Such test-driven development (TDD, cf. [1]) is reported to be successfully implemented—even in large-scale companies like Google [2], which has implemented a whole department dedicated to take care of the required tools and processes and to coach the product development teams.

While the test strategy towards TDD is described well in literature, the specific situation in a given company needs to be analyzed first before concrete actions can be initiated to improve a company's concrete test, integration, and deployment system. Therefore, the very first step is the identification of the company's status quo in terms of listing and presenting their current test initiatives. Therefore, related approaches are described briefly. For further information, we refer the reader to Ståhl and Bosch [3] and Nilsson et al. [4].

In [5], Stolberg describes an approach to compare the situation before and after applying Fowler's checklist for introducing continuous integration. However, an intuitive and integrated visualization in terms mapping the company's current situation for testing a product is not proposed.

Guidelines for monitoring systems for software builds are provided and discussed by Downs et al. [6]. While they are focusing on how to utilize results from broken builds during the software development, a visualization chart as proposed by the "CIViT model" is missing.

Experience reports from using existing tools like CruiseControl to realize continuous integration and to get feedback in an automated manner from software builds are presented in different works like Sturdevant (cf. [7]) and by Kim et al. (cf. [8, 9]).

Another experience report provided by Hoffman et al. in [10] outlines the use of the commercially supported toolchain around CMake in a governmentally driven research lab. In contrast to the CIViT model, they are focusing on the technical introduction of the toolchain and not on deficiencies of current test strategies and quality assurance processes.

As evident from literature, an intuitive approach that helps the involved stakeholders to explicate the current situation in their company regarding processes, methods, and tooling to unveil bottlenecks is not available. Thus, our work presented in [4] systematically evaluated for the first time in a large-scale setting the needs and requirements for an intuitive visualization approach of a company's current situation of testing and integration efforts. Furthermore, that report evaluates the applicability of the CIViT model with five development sites from four companies that are partners of the Software Center. This article now gives an overview about the CIViT model and serves as a practitioner's guide.

8.3 Continuous Integration Visualization Technique

The Continuous Integration Visualization Technique (CIViT) model is concerned with four types of testing: new functionality, legacy functionality, quality attributes, and edge cases. New functionality testing refers to testing the functionality of the system currently under development. Legacy functionality testing refers to the functionality that was already built and to ensure it still operates according to its specification. Quality attributes testing refers to, e.g., performance, reliability, safety, and security and intends to ensure that the system continues to satisfy the specified quality requirements. Lastly, edge case testing refers to testing unlikely or weird situations that, often, originate from faults that slipped through to customers and that were discovered after significant investigative effort. This type of testing is not often mentioned in the literature, but from the interviews in this study, it became obvious that this is an important type of testing.

Figure 8.1 outlines four squares forming a bigger square. The "F" represents new functionality, the "L" represents legacy functionality, the "Q" represents quality attributes, and the "E" represents edge cases. Each of these squares can have one of three colors: red, orange, or green. The color of the square indicates the level of coverage of test cases in the specific square. The mapping between coverage and color-coding is shown in the upper right in Fig. 8.1. The line surrounding the four squares indicates the level of automation and again uses the same color-coding. The mapping between colors and level of automation is shown in the lower right part of the figure.

The CIViT model has two dimensions: **scope** and **periodicity**. The scope of testing refers to the segment of the overall system that is being tested. The scope of

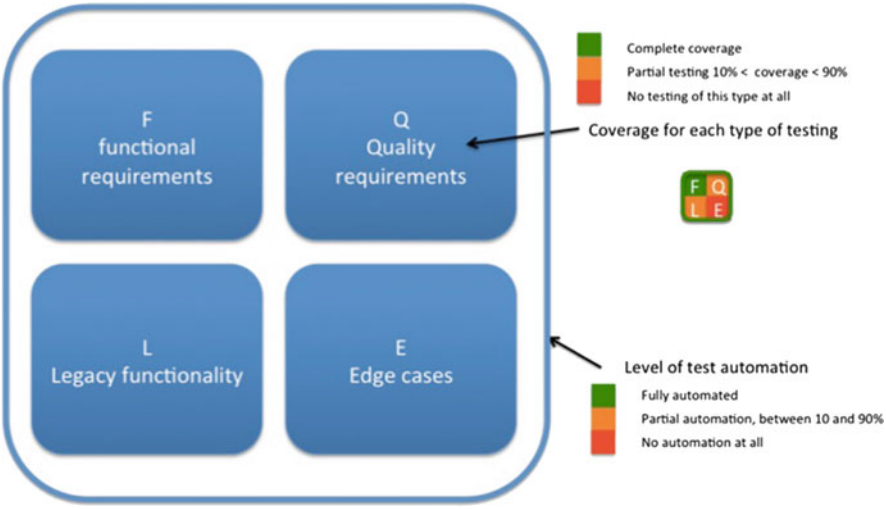


Fig. 8.1 Four types of testing in the CIViT model

testing dimension is divided into five levels: component, subsystem, partial product, full product, on-site release, and customer-site release. Below we describe each scope in more detail:

Component or module refers to a part of the system that would usually be the scope of an individual engineer or a small team and typically involves unit testing.

Subsystem refers to the scope of responsibility for a team or a small set of teams, and the types of test cases are broader in the area of covered functionality and less white box than at the previous level.

Partial product refers to system level testing in which some parts of the mechanics and hardware of the system are available and other parts are simulated. Typically, test rigs that combine the most important aspects in a structure that allows for testing the primary functional and quality requirements.

Product refers to the full product with all parts present, including mechanics, hardware, and all software. The challenge with product-level testing is that the cost of providing the full product is often quite high, and in cases where hardware and mechanics are developed in parallel with the software, the full product typically becomes available late in the development process.

Release refers to the full product for all aspects that are of importance to the customer and is concerned with the completeness of testing, including edge cases and all quality attributes of secondary priority, ensuring the expected functionality and quality at the customer site.

Customer refers to installing the system or product at the customer site and performing testing activities to ensure the correct operation of the system in the context of the customer.

The second dimension of the CIViT model is concerned with the periodicity of testing. We define periodicity as the combination of the frequency of a testing activity and the time between the start of the testing activity and the availability of feedback from that testing activity. We identify three levels of periodicity: "in the development workflow" (**minutes** and **hours**), "disrupting the development workflow" (**days** and **weeks**), and "outside the development workflow" (**months** and **once** per release).

While feedback within days or one or a few weeks was perceived as relatively good periodicity among the companies, this would still often be experienced as disruptive to the development workflow. Typically, the team working on a feature has moved on to other tasks, and feedback about errors returned after days or weeks requires the team to stop their current work and return to the previous work, i.e., a context switch, make the change, submit, and then return to the task they were working on before the returned feedback.

The even longer periodicity, i.e., months or once per release, often results in high-level system errors that usually are quite complex and difficult to analyze and understand where and what the actual root causes are. In this case, any defects that are found are typically resolved by engineers different from those that introduced these defects.

The CIViT model aims to indicate the order of magnitude of the feedback loop, rather than providing the exact length. For instance, "hours" indicates from one to a

Fig. 8.2 An example instantiation of the CIViT model from one of the participating companies

small number of hours, but clearly less than a day, and similarly regarding days, weeks, and months.

In Fig. 8.2 an example instantiation is shown. Each composite square indicates a distinct point of testing in the end-to-end testing activities. At each point of testing, one or more of the four types of testing is conducted. For each type of testing, the coverage is indicated by the color-coding introduced earlier. In addition, the line around the four smaller squares indicates the level of automation for testing at that point in the quality assurance process.

8.4 Applying CIViT Model: An Experience Report

In this section, we report about applying the CIViT model to one of the companies from the Software Center. The company we are presenting here operates in the telecom space and is in the process of transitioning from biannual releases to more deployment of software, i.e., at the end of every agile sprint. The company has used the CIViT model to outline its current state of testing along the software development lifecycle and, based on the current state analysis, identify and prioritize the most important improvements to implement.

In Fig. 8.2, we show the CIViT model from this company. As this company is quite advanced compared to other case study companies where we have applied the model, one of the key observations is that code from developers and teams becomes

part of the product baseline very quickly, typically within minutes. This is visible in the vertical set of squares at the left side of the figure.

The second area of interest is the four squares next to each other horizontally at the full product level. These squares indicate increasing scopes of testing that is conducted less and less frequently:

- The most right square in the lowest corner is an automated test activity that runs every time new code is checked in to the developer base line. The duration of the testing is less than a minute. The purpose of this testing effort is to remove the most obvious errors.
- The square above the aforementioned square is an automated test suite for the team base line and tests the checked in functionality in the broader team context. The test runs for a couple of minutes and is intended primarily for functional requirements.
- The highest square at the right is the automated test suite at the base line for the product. Any new code first needs to pass this test suite before becoming part of the base line of the product. In practice, the test suite focuses on functional requirements but also includes some legacy functionality and rudimentary quality attribute test cases.
- The next square to the left is an automated test suite that runs every two hours (assuming new code has been checked in). The duration of the testing effort is two hours as well. The R&D organization selects the highest priority tests that fit in a 2-hour test window. The tests cover all four types of testing but still focus more on functionality and legacy.
- The third square is nightly testing effort that takes about 10 h to execute. Whereas the other testing efforts primarily focus on functionality and legacy, the nightly testing has a stronger focus on the quality attributes of the system including performance, robustness, throughput, etc. Similar to earlier, the R&D organization selects the highest priority test cases for the nightly test.
- The most left square is a very elaborate test suite that runs over the weekend and takes around 50 h to complete. The focus of testing is especially on those qualities and configurations that can only be tested over longer testing periods such as stability, memory leaks, and other more subtle errors.

The third area of interest is the testing activities at the release and customer level. The organization has a separate release testing team that takes the product baseline at the end of every agile sprint and performs testing on it. For the version selected for release to customers, the release testing organization performs a very elaborate testing effort, followed by a testing effort at the customer site in a live telecom network. When these testing efforts have been successfully concluded, the product is made available to all customers. Whereas the testing by the R&D organization is entirely automated, the release testing organization performs virtually all its testing in a manual capacity.

8.5 Evaluating the CIViT Model

The CIViT model was developed in response to a set of challenges that we identified as part of our work with the Software Center companies. In an earlier paper [4], we discussed these challenges and resolution that the CIViT model provides. In this section, we provide a brief overview of these challenges and the way that the CIViT model addresses these.

No end-to-end overview of testing in companies: The primary concern that we identified in our research is that few companies have a comprehensive overview of all testing activities between a front-line developer and the deployed system at a customer. The CIViT model was explicitly developed to provide this overview. As a first validation step, together with representatives from each company, we developed CIViT models for each participating company. As a second validation step, we used the model for each company to identify what testing activities in their model that they would like to focus on to improve. Each company selected a specific box in their model and explicated in what way they aimed to improve the selected testing activities, for example, by increasing periodicity from, e.g., month to week or by increasing scope from, e.g., subsystem to partial product.

Significant duplicate testing efforts: During the elaboration of end-to-end testing activities, representatives from different groups in the company were brought together. As part of the workshops, it became abundantly clear that different groups performed significant duplicate effort due to the lack of understanding of what was tested in earlier stages. The overview provided by the CIViT model enables useful discussions that reveal what type and quality of testing that are performed within the settings. Our research shows that this is helpful to identify unintended and undesired duplicate testing efforts, as well as to ensure that sufficient testing efforts are in place at the various levels of the end-to-end process.

Slow feedback loops: In a similar way, the CIViT model both visualizes directly the periodicity of the involved testing activities and consequently reveals their feedback loops in the settings and enables useful discussions about what would be reasonable and desired times of feedback loops within the end-to-end process of testing activities.

Late testing of quality attributes: Several of the companies involved in the research raised as a concern that quality requirement violations were often identified late in the testing process, were very costly to correct at that stage, and caused a significant lack of predictability. The CIViT model also directly visualizes what different types of testing that are dealt with in the involved testing activities. For example, the study shows that this helps to reveal to what extent the testing of quality attributes, e.g., performance and robustness, takes place and when.

Ad hoc, tactical improvement efforts: When we reviewed the improvements in testing performed at the companies before the start of our research, it became clear that improvements were selected in a rather ad hoc fashion. We identified several cases where improvements were clearly suboptimal in that alternative improvements would have resulted in significantly more relevant benefits. Based on the overview that the CIViT model provides, it also enables useful discussions of the

testing activities that are performed within the settings regarding what areas would be suitable to improve and how. This helps the companies to move away from the typical ad hoc approach towards improvement efforts and have a better understanding of the end-to-end verification process and the key issues when they make decisions about what to do and how.

Summary and Conclusions

In this article, we have described the CIViT model, which is an intuitive visualization technique to explicate a company's current situation regarding testing and integration efforts. Therefore, we have identified a two-dimensional chart that relates the dimensions periodicity from minutes based, over hourly, weekly, monthly to once per release on its *x*-axis and scope of testing on the *y*-axis ranging from software components only to full-scale product.

On the intersection points in this chart, boxes are placed that describe the methods and scopes are applied for the considered combination of periodicity and scope of testing. These boxed describe in a compact manner the level of automation for the considered dimensions: functional code, legacy code, edge cases, and quality aspects (nonfunctional requirements). Thus, it is intuitively and clearly visible, where the bottlenecks towards continuous testing and continuous integration arise concerning automation.

Furthermore, it is evident from the overall placement of these boxes, how "well" the testing and integration process performs: The longer it takes to integrate and test a more complete product, the slower is the final deployment to the market. Thus, the product owners and the various process owners for the functional development and feature testing can easily spot those related boxes, where an improvement regarding information flow, for instance, is required to reduce the integration and testing effort.

The CIViT model addresses several challenges identified in earlier work [4]. These challenges include the lack of an end-to-end overview of testing activities in the company, the duplication of testing efforts, the slow feedback on new code, the late testing of quality attributes, and the ad hoc test improvement activities engaged on by many companies. We have evaluated the CIViT model with several companies, and we have seen significant improvements in the aforementioned areas. The ability to visualize testing activities in an intuitive, illustrative model that summarizes the key areas provides significantly improved understanding and an excellent basis for identifying the most important improvement areas.

As future work, domain-specific guidelines regarding improvement initiatives are of clear interest. Thus, domain expert would be able to learn from experiences of companies in similar or related domains when considering to adapt and evolve the own processes, methods, and development and testing tooling.

Acknowledgments The authors would like to thank all engineers, testers, and managers who were involved in our work towards the CIViT model.

References

1. Beck, K.: Test Driven Development: By Example. Addison-Wesley Professional, Boston (2002)
2. Whittaker, J.A., Arbon, C., Carollo, J.: How Google Tests Software. Addison-Wesley Professional, Boston (2012)
3. Ståhl, D., Bosch, J.: Modeling continuous integration practice differences in industry software development. J. Syst. Softw. **87**, 48–59 (2014)
4. Nilsson, A., Bosch, J., Berger, C.: Visualizing testing activities to support continuous integration: A multiple case study. In: Proceedings of the 15th International Conference on Agile Software Development (2014)
5. Stolberg, S.: Enabling agile testing through continuous integration. In: Proceedings of the Agile Conference, pp. 369–374 (2009)
6. Downs, J., Hosking, J., Plimmer, B.: Status communication in agile software teams: A case study. In: Proceedings of the Fifth International Conference on Software Engineering Advances, pp. 82–87 (2010)
7. Sturdevant, K.: Cruisin' and Chillin': Testing the java-based distributed ground data system 'Chill' with CruiseControl. In: Aerospace Conference, pp. 1–8 (2007)
8. Kim, E.H., Na, J.C., Ryoo, S.M.: Implementing an effective test automation framework. In: Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference, pp. 534–538 (2009)
9. Kim, E.H., Na, J.C., Ryoo, S.M.: Test automation framework for implementing continuous integration. In: Proceedings of the Sixth International Conference on Information Technology: New Generations, pp. 784–789 (2009)
10. Hoffman, B., Cole, D., Vines, J.: Software process for rapid development of HPC software using CMake. In: Proceedings of the DoD High Performance Computing Modernization Program Users Group Conference, pp. 378–382 (2009)

Chapter 9
Continuous Integration Flows

Daniel Ståhl and Jan Bosch

Abstract While the agile practice of continuous integration has gained increasing traction in industry since its popularization in the 1990s, there is considerable diversity in terms of actual implementation. The term has been used to describe what may in practice be described as rather different practices, with subsequently varying outcomes. This diversity, typically camouflaged by common terminology, not only prevents effective comparison and therefore learning from industry cases but also hinders practitioners in making informed choices as to how continuous integration is best implemented in their particular context. To facilitate analysis and experience exchange, we present a descriptive model of automated software integration flows. Then, helping software professionals with their ability to proactively and consciously build integration system suitable to their needs, we propose an iterative method for integration flow design.

9.1 Defining Continuous Integration

Continuous integration is one of the most popular and commonly aspired agile practices in the software development industry. Unlike other practices, such as retrospectives, burndown charts, or backlogs, it is one whose implementation itself requires considerable software engineering efforts. Furthermore, in literature as well as among practitioners, continuous integration is quite loosely defined, with ample room for interpretation concerning its implementation and little consensus with regard to its scope, characteristics or effects. To illustrate this, we can consider the often-quoted definition of continuous integration put forward by Martin Fowler:

D. Ståhl (✉)
Ericsson AB, Linköping, Sweden
e-mail: daniel.stahl@ericsson.com

J. Bosch
Chalmers University of Technology, Gothenburg, Sweden
e-mail: Jan@JanBosch.com

J. Bosch (ed.), *Continuous Software Engineering*,
DOI 10.1007/978-3-319-11283-1_9

> Continuous Integration is a software development practice where members of a team integrate their work frequently; usually each person integrates at least daily—leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible [1].

Such a definition is ostensibly straightforward and clear-cut, but once that surface is scratched, numerous questions arise. What if there is more than one team? How and when do they integrate with one another? What is the scope of the automated testing? What is it the team members are integrating with: a development branch, a release branch, or perhaps a feature branch? How are results of the automated builds fed back to the interested parties, and who are they? How is component integration handled in a modularized product architecture? These are just some of the questions with which professionals in the industry wrestle with on a daily basis—and come up with different answers to. Indeed, when we study continuous integration as it is employed in practice, we find that not only are perceptions as to its effects wildly different, or even diametrically opposed, but the actual continuous integration systems themselves also diverge at a large number of variation points. Consequently, the term continuous integration is in practice used to describe a disparate family of measures which revolve around the concepts of automation and frequent repetition.

In this light the tendency of business leaders and managers to proclaim either that they have adopted continuous integration, or declaring that it shall be adopted, is highly intriguing. Given the diversity present in industry, simply saying that one is to implement continuous integration without defining in any greater detail what that is understood to mean, or what one expects to achieve by doing so, can be likened to rolling a die and hoping that something useful will turn up, that is, if we assume that any effects of continuous integration are positive effects: in reality, software integration systems can have severe negative consequences for the development effort, whether they are continuous or not, particularly in large-scale projects. In other words, as we roll that die, we also hope that we won't accidentally implement anything harmful. Those of us who would rather design our integration systems consciously and proactively, rather than gamble, will look for methods for designing continuous integration to suit best further our goals, given our particular circumstances.

To get there, let us first return to the question of what continuous integration is, and how it is that it adds value. We already learned how people place different meanings in the term and in reality perform very different activities. We also find that generally speaking, whichever activities they do perform, they're happy about them being "continuous" and derive value from that, while the continuous integration system itself is of little interest except to those who are tasked with developing and maintaining it. In other words, continuous integration as it is employed in industry becomes much more easily comprehensible if we think of it not as a fixed and well-defined practice, with some expected benefits. Instead, it should be thought of as an enhancer of other value-adding activities which we may or may

not already be performing in our software development projects: unit testing, code analysis, performance testing, system testing, packaging deployment, and so on. These are all beneficial or even necessary activities which may potentially be enhanced by being made "continuous," that is, by automating them and striving to perform them more frequently.

To exemplify, having a unit test suite may be a great asset in that it allows developers to locally verify any changes they make, repeatedly executing that test suite on the project's main branch whenever changes are checked in and then broadcasting the results for all to see can significantly increase its value. That value can still only be as great as the quality and scope of the test suite, however. A suite of vapid, meaningless tests will never be useful to anyone, no matter how "continuously" one executes it. Again, it is not continuous integration that adds value; it is the activities we apply it to that are enhanced by it.

One might object to this view by arguing that this only addresses the "automated build" side of the practice—the essential practice is that of integrating often and frequently and that in theory this can be accomplished without any automation at all. While this is technically true, what we find in practice is quite different: one of few common denominators of industry implementations is automation of some set of activities following that integration, and through investigation and interviews with software professionals, one finds that it is these continuously performed activities that provide value.

This is not surprising if one would imagine a development project where developers integrated ever so frequently, but no feedback as to the functionality or quality of the resulting source code revision was provided; the benefits derived from this integration would be limited. This should not be taken to mean that the act of frequent integration is irrelevant. On the contrary, it is a prerequisite for frequently executing the activities from which the development project benefits.

From this perspective of continuous integration as an enhancer of other activities, of their own intrinsic value, the fact that different organizations and individuals experience disparate effects and report varied implementations can easily be explained: different sets of activities have been included in the context of continuous integration, with varying internal relationships between them.

This does not imply that how we construct our continuous integration systems is without consequence, however. As experiences differ, with some being more positive than others, what we make "continuous" and how we do it clearly matters, prompting us to investigate any relationships between continuous integration flavors and their outcome. This requires a less superficial understanding of the practice: one that goes beyond the buzz word level and allows us to accurately and unambiguously describe both concrete and theoretical software integration systems so that they may be documented, compared, and evaluated an essential capacity if we are to draw any conclusions from past experiences and evolve as an industry.

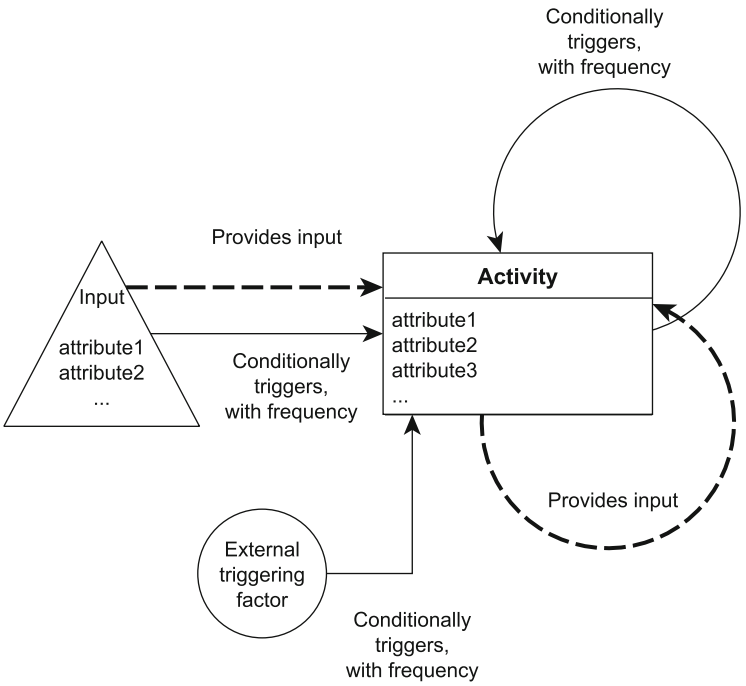


Fig. 9.1 Software integration system meta-model

9.2 Modeling Software Integration Systems

To achieve a more profound understanding of the actual individual case, we have developed a method of modeling software integration systems. It is based on the concept of interconnected and interdependent activities, or "builds," as a directed acyclic graph [2] and is designed to cover all continuous integration variation points uncovered in literature [3]. Its meta-model is shown in Fig. 9.1. Activities, software input, and external triggering factors are represented as nodes, while triggering and input relationships constitute edges.

The model has subsequently been applied multiple times to software development projects in the industry, allowing us to build a knowledge base of existing implementations. In combination with testimonies of professionals involved in those projects, not only does this afford us a much improved understanding of actual continuous integration implementations and their effects but has also turned out to be a valuable tool for the participating engineers to better understanding their own systems: once it has been mapped out and the modeled representation is constructed, the paths along which software "flows" between activities become discernible. In short, software revisions or artifacts traverse, or flow through, the graph of automated activities, and as they are subjected to these activities, our knowledge about and confidence in the software increase. In applying the model to

industry cases, our experience is that the professionals responsible for designing and maintaining the studied continuous integration systems are not used to thinking of them in these terms but soon adopt the concept as it highlights concerns in the existing systems. For this reason, we use the term *integration flows* to describe these systems of interconnected automated activities processing software revisions.

Analysis of the data gathered through these case studies further reveals a set of guidelines, or best practices, which increase the value derived from integration flows.

- **Comprehensive Activities**. As discussed above, various activities can be placed in a continuous integration context, and as a general rule, their benefits to the development project increase as a consequence. What we see in practice, however, is that most practitioners only perform a small subset of these activities—they might, for instance, execute unit test suites, but no system level tests—causing them to miss out. The rule of thumb here is that the more of the project's activities can be made continuous, the greater the rewards.
- **Effective Communication**. Even the most ambitious integration flow falls short of its potential if the results it produces are not clearly communicated to its stakeholders, such as developers, managers, and product owners. Different stakeholders may require different information—the developer might primarily be interested in whether the latest check-in has passed all tests, whereas a project manager rather looks for acceptance test results in a certain release track or which product versions have been deployed to which customers. All this is information which a continuous integration flow may potentially provide, but in practice we see examples where even if such data is produced it's not accessible by those who would benefit from it or it is provided in such a cryptic format that only a handful of specialized individuals are able to make sense of it. Consequently, communication is an important aspect to consider when designing continuous integration flows: who are the stakeholders, what type of information do they require, and on which format do they need it?
- **Immediacy**. Not all continuous integration flows we see in industry are what one might consider continuous from the perspective of the individual developer. While new revisions of the product are repeatedly built and tested many times a day, the frequency and speed at which the developer gets feedback are very low. Large projects in particular are susceptible to this phenomenon: even though the continuous integration flow builds as quickly and frequently as possible, there are simply too many project members for all of them to be granted the instant feedback they seek on their individual changes.
- **Accuracy**. As previously discussed, software revisions or artifacts "flow" through continuous integration systems, whereupon our level of confidence in them increases as the automated activities successfully process them. This requires that we know well the identity of the artifact being processed: if this is uncertain or vague, the concept of successively growing confidence becomes meaningless. We see examples of this in the industry, where an automated activity is triggered by the successful conclusion of an upstream activity, but

does not consume its output. Instead, it checks out its own source code revision. The consequence of this is that what is processes is now "an artifact revision which has passed such and such tests," but rather "an artifact close in time, and possibly identical, to one which has passed such and such tests." In other words, it is important to keep accurate track of the software revisions and artifacts produced—arguably a crucial practice regardless of one's integration strategy.

- **Lucidity**. Once a continuous integration flow is in place, it's not necessarily understood by all the project members or even the engineers responsible for designing and maintaining it, exactly how its activities relate to each other, and which paths a given software artifact may take in traversing it. In studying industry cases, we find that in projects of all sizes, developers can be ignorant of what happens to a software revision after they check it in and that they consequently don't know what feedback they can expect or where to look for it, thereby being deprived of its potential benefits. Even those responsible for the integration flows can in some cases find it difficult to answer questions such as which tests must have been passed for an artifact to reach a certain point in the flow, or what happens downstream of a given activity depending on its outcome. Feedback received in such situations shows us that building a model of the integration flow is a helpful exercise to make its behavior visibly clear where uncertainty exists.
- **Appropriate Dimensioning**. Our final guideline is that of dimensioning of the integration flow. It has been pointed out in literature [4, 5] that continuous integration can be difficult to scale. In our case studies, we have been able to confirm this—indeed, when some of the problems discussed above emerge in large projects, they seem to be caused to some extent by the failure to ensure sufficient capacity in the continuous integration flow. This is because larger projects tend to imply not only larger products and longer build and test times but also a higher pace of changes or software artifacts to be integrated. In addition, the larger the project, the greater the consequences when difficulties arise, exacerbating the problem. What we mean by integration flow capacity is thus the amount of software artifacts that can be handled without adverse effects growing out of hand—effects which can include build fragility, queuing, and long feedback times. Finally, we also find that an integration flow can be overdimensioned: being too modularized, with too many parallel paths in the flow, can cause overhead and unnecessarily high maintenance costs.

Based on our research and experience, we consider all of these guidelines important in maximizing one's benefits from continuous integration flows. A few factors set the question of dimensioning apart from the others, however. First, the problem of scaling continuous integration is inherent in the practice, as opposed to the relatively superficial concerns of, e.g., how to communicate the results or how to make the flow more comprehensive or easily accessible. Also, while these latter aspects can be addressed in existing systems with relative ease, we find that tuning the capacity of the integration flow can be both costly and fraught with risk for the project. This is because the flow capacity is tightly linked to its modularity: as a rule

of thumb, the more parallel paths in the integration flow, the greater its capacity. This, in turn, is linked to the modularity of the product itself. A highly modular product architecture can easily be translated into a modular, high capacity integration flow. The integration of a monolithic product, on the other hand, is difficult to parallelize and consequently to scale up. Conversely, an overly modular product architecture soon becomes unwieldy and difficult to develop. This leads us to the conclusion that any ability to foresee the capacity needs of a given project, enabling proactive appropriate dimensioning of its integration flow, would be of great value.

9.3 Proactive Analysis

To devise a method of proactive analysis of software integration systems, we return to our modeling technique: if we are able to determine the capacity of a modeled integration flow—real or hypothetical—and compare that to a project's estimated capacity requirements, this will tell us whether that integration flow is appropriately dimensioned or not. To this end, we introduce the concept of artifacts per potential executions (APPE) ratio. The APPE ratio signifies how many software artifacts (e.g., product revisions) need to be handled per execution of an automated activity, if that activity would execute at maximum frequency. To illustrate this, imagine an automated activity which compiles a product component. Due to its execution time, it's able to run a maximum of 20 times a day. If, on an average day, five new revisions of the component are pushed by its developers, there are four potential executions for every artifact (i.e., an APPE ratio of 0.25). When this ratio approaches one, however, one expects to see intermittent queuing and/or batching of artifacts, as developer pushes more frequently coincide in time. If the APPE ratio is much higher than one, on the other hand, either because the number of artifacts is very high or because the number of executions is very low, increasingly negative effects manifest: large numbers of artifacts are batched in single activity executions, leading to more integration faults as well as difficulty in analyzing and recovering from those faults.

By applying this capacity estimation method to actual integration flows, its pain points can be identified so that improvement efforts can target the parts of the flow that will provide the greatest benefit. Meanwhile, applying it to hypothetical integration flows allows for proactive design of one's integration flow in order to ensure suitable capacity for the project's estimated needs. We propose that this capacity estimation is applied as part of an iterative integration flow design process, depicted in Fig. 9.2.

The first step is to create an initial model of the planned integration flow. This is not intended to be the final version, so it doesn't have to be perfect. A good starting point is to list the automated activities (e.g., compilation, linking, various tests, packaging, and deployment) one would like to include in the flow, estimate their duration, and connect them as shown in Fig. 9.1. The resource requirements for realizing this flow must then be estimated: what competencies and manpower

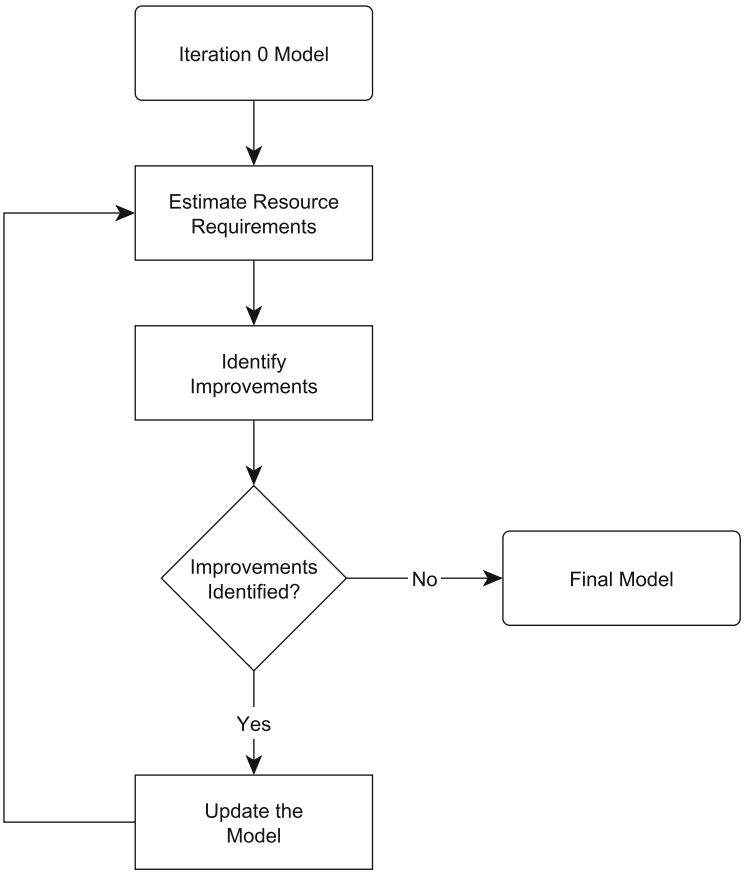


Fig. 9.2 An iterative design process for software integration flows

would be needed to implement and maintain the flow and what hardware would be needed to execute the activities.

Having come this far, it's time to look for needed improvements. There are mainly three tools at our disposal to identify such improvements. First, are the resource requirements acceptable? Perhaps the projected integration flow would be too expensive to implement or require too much hardware? Second, does it conform to the integration flow guidelines of comprehensive activities, effective communication, immediacy, accuracy, and lucidity? Finally, is the capacity sufficient? By estimating the pace of development, the APPE ratio of the automated activities can be calculated and a qualified decision made as to whether the project's needs will be met.

If needed improvements have been identified, the model is revised by, e.g., changing the activity relationships or their characteristics, and a new iteration of the process begins, with resource requirements estimation. If not, the process is complete, and work can begin on implementing the modeled integration flow.

To exemplify, it might be found that the projected duration of an activity is unacceptable. The model is revised, with the duration being shortened to an acceptable level. In the next iteration of the process, it is concluded that additional hardware is required in order to achieve this shorter duration. Whether this extra cost is acceptable or prompts additional changes to the integration flow is then discussed in the subsequent improvement finding step.

We recommend that only a few improvements are addressed each iteration of the process: it is better to improve the model in small increments than to try to fix everything at once and make too drastic changes. It should also be pointed out that this exercise might conclude that the product architecture needs to be changed to achieve a better integration flow. In other words, integration engineers must be allowed to influence architecture—not just be presented with a fait accompli and try to make the best of it, as is typically the case in the industry projects we have witnessed and been part of.

In conclusion, our understanding of continuous integration is rapidly improving and with it our ability to make deliberate, proactive, and conscious choices to maximize the value we derive from the practice. Not only can we establish that some implementations add more value than others, but we have guidelines for reproducing their success. And not only do we find that continuous integration can be problematic, but we can foresee those problems beforehand and prescribe solutions to avoid them. Looking ahead, we anticipate that these tools and insights will be honed further, as our practical experience from applying them increases even further.

References

1. Continuous Integration. http://martinfowler.com/articles/continuousIntegration.html. Accessed 10 Dec 2013
2. Beaumont, O., Bonichon, N., Courtès, L., Hanin, X., Dolstra, E.: Mixed data-parallel scheduling for distributed continuous integration. In: 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum, pp. 91–98. IEEE, Shanghai (2012)
3. Ståhl, D., Bosch, J.: Modeling continuous integration practice differences in industry software development. J. Syst. Softw. **87**, 48–59 (2014)
4. Rogers, R.O.: Scaling continuous integration. In: 6th International Conference on Extreme Programming and Agile Processes in Software Engineering, pp. 68–76. Springer, Sheffield (2005)
5. Roberts, M.: Enterprise continuous integration using binary dependencies. In: 5th International Conference on Extreme Programming and Agile Processes in Software Engineering, pp. 194–201. Springer, Garmisch-Partenkirchen (2004)

Chapter 10
Towards Continuous Integration for Cyber-Physical Systems on the Example of Self-Driving Miniature Cars

Christian Berger

Abstract Today's consumer life is already pervasively supported by visible and unnoticeable technology. We are consuming information flows, contributing within social webs, and integrating our virtual communities into an interconnected lifestyle. This interconnected and assisted way of living is realized by various products ranging from consumer electronics products like smartphones and wearable computing up to safety-critical systems like intelligent cars, which aim for unnoticeably protecting the user and its surroundings in critical situations. And at the end of this decade, the technology of a self-driving car is reported to be available for consumers enabling various opportunities for new businesses.

From consumer-level technology like smartphones, smart TVs, or laptops, users are used to feature extensions and evolution over time by having automated application and operating system updates. Thus, further system features are continuously rolled out on a large user base enabling new use cases. Nowadays, the digitally connected lifestyle integrates components like wearable computing and smart mobility, where an OEM could hardly anticipate the nearly limitless variety of complex combinations.

The trend of a continuously evolving user-experience in terms of new features and functionalities puts further challenges, requirements, and constraints on a system provider to maintain the expected high quality of the product and in the future of the interconnected and integrated product network.

This article presents the design of a simulation-based testing and integration approach for cyber-physical systems by using self-driving miniature cars as the running example.

C. Berger (✉)
Department of Computer Science and Engineering, Chalmers University of Gothenburg, Gothenburg, Sweden
e-mail: christian.berger@chalmers.se

J. Bosch (ed.), *Continuous Software Engineering*,
DOI 10.1007/978-3-319-11283-1_10

10.1 Introduction

Self-driving vehicles are expected to be available for customers by the end of this decade from several major vehicle original equipment manufacturers (OEMs) [1]. Thus, the technology has significantly improved from vehicles reportedly addressing safer travelling [2] over the recent robotic challenges like the DARPA Grand Challenges from 2004, 2005, and 2007, as well as the Grand Cooperative Driving Challenge from 2011. These practical showcases were mainly focusing on engineering challenges (cf. [3–6]) to demonstrate the technological state of the art.

While this technology continuously finds its way into today's cars as adaptive cruise control, lane departure warning, or self-parking assistants, testing and integrating these software systems with the hardware like sensors and actuators are challenging. Some reasons for that are the increasing complexity of the traffic scenarios addressed by such systems, ensuring the repeatability of the tests to evaluate the fulfillment of the expected behavior, and the time-consuming test execution on proving grounds or in field tests.

In this regard, virtual test environments do not only allow interactive validations of such algorithms on the developers' and testers' desk; additionally, such environments can support a continuous integration to tackle the increasing testing load caused by growing product families and more scenarios to be supported by such systems. As a running example, this article uses an autonomous box-parking system to describe the design of such a simulation-based virtual test environment to enable continuous integration for cyber-physical systems (CPS).

The rest of the article is structured as follows: In Sect. 10.2, the functionality of autonomous box parking used as running example in this article is outlined comprising the sensor layout and one possible state machine that is used to realize such functionality. In Sect. 10.3, the design of a simulation-based testing and integration approach for cyber-physical systems on the example of a self-driving miniature car is outlined. The article concludes in Sect. 10.4.

10.2 Autonomous Box Parking

In the following, the exemplary traffic scenario for parking autonomously between several cars is depicted alongside a sensor layout and a possible state machine realizing the parking behavior.

10.2.1 Exemplary Parking Scenario

A "box"-parking scenario as depicted in Fig. 10.1. A self-driving and autonomously parking car is initially located at the origin of the coordinate system at the bottom of

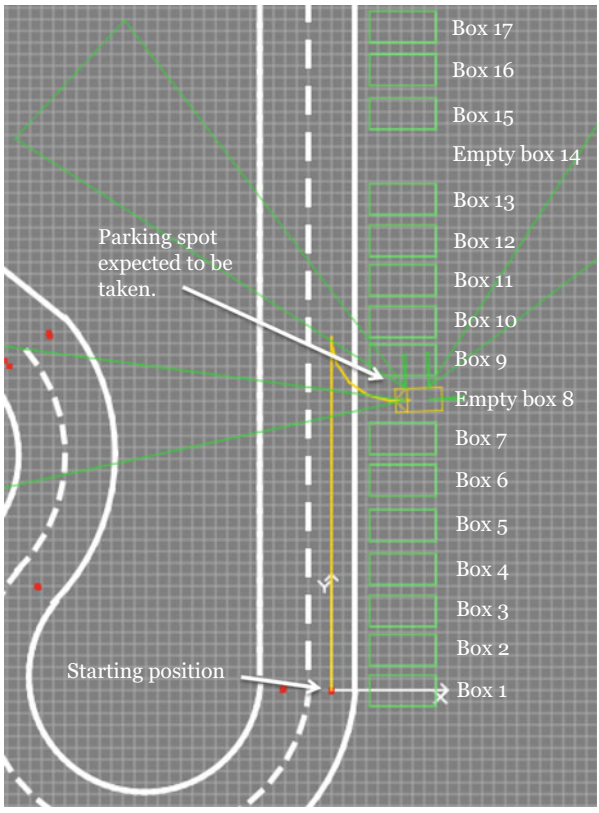


Fig. 10.1 Exemplary box-parking scenario

the image heading towards to the upper part of the image. The car is driving on the right-hand side of the straight road following the lane markings while continuously observing its right-hand side to find a parking spot. Once it has found a spot, which is sufficiently wide enough, the car stops, steers at maximum to the right, and drives backwards into the parking spot, where it comes to a stop.

10.2.2 Perceiving the Surroundings

A self-driving car exhibits several sensors to perceive its surroundings [5]. Figure 10.2 depicts such a sensor layout, which has proven to be successful during an international competition for self-driving miniature cars for the vehicle "Meili" from Chalmers University and University of Gothenburg that won the Junior Edition in 2013 [7]. The competition required participants to develop a vehicular robotic platform in 1/10 scale, which is able to autonomously follow lane markings, overtaking stationary and dynamic obstacles, yielding right of way at intersections, and realizing sideways parking.

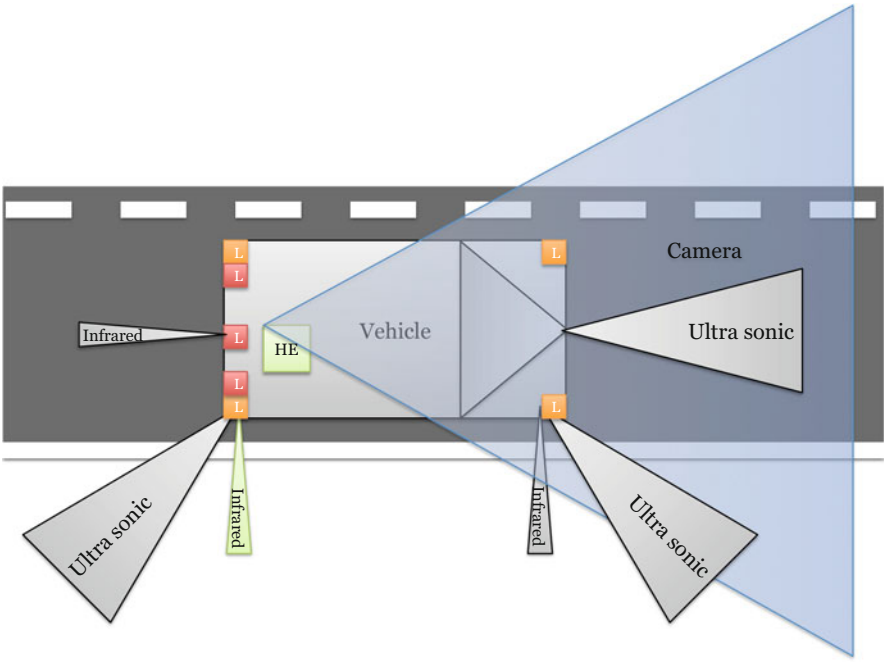


Fig. 10.2 Experimental sensor layout to find a parking spot consisting of a sensor to determine the travelled path (HE) and a distance-based sensor to measure the distance to obstacles to the *right-hand side* (infrared) highlighted in *green*

For the functionality for the autonomous box parking within a self-driving car, we are focusing only on those aspects relevant for finding a parking spot that is sufficiently wide enough while excluding further parts from a self-driving car like image processing for lane following for the sake of clarity.

Autonomous self-parking as described in this paper considers a box-parking approach, where the car is finally oriented in an orthogonal direction. For measuring the size of a parking gap, the travelled distance over time together with information about distances between the car and its right-hand side is required. In Fig. 10.2, the distance sensor mounted at the rear/right corner and pointing towards the vehicle's right-hand side is highlighted. Using this sensor in comparison with another sensor that is mounted further at the front side of the vehicle has the advantage that the car already passed a parking spot, which is sufficiently wide enough, and thus, further correcting maneuvers are not required to put the vehicle in a valid starting position for the subsequent parking trajectory. Furthermore, a hall effect sensor (labelled HE) is used to determine the travelled distance over time. Combining both allows for determining the size of possible parking gaps.

Figure 10.3 shows the distances to obstacles on the vehicle's right-hand side returned from the rear/right distance sensor over time during a simulation run for the scenario depicted in Fig. 10.1: whenever the distance sensor does not perceive

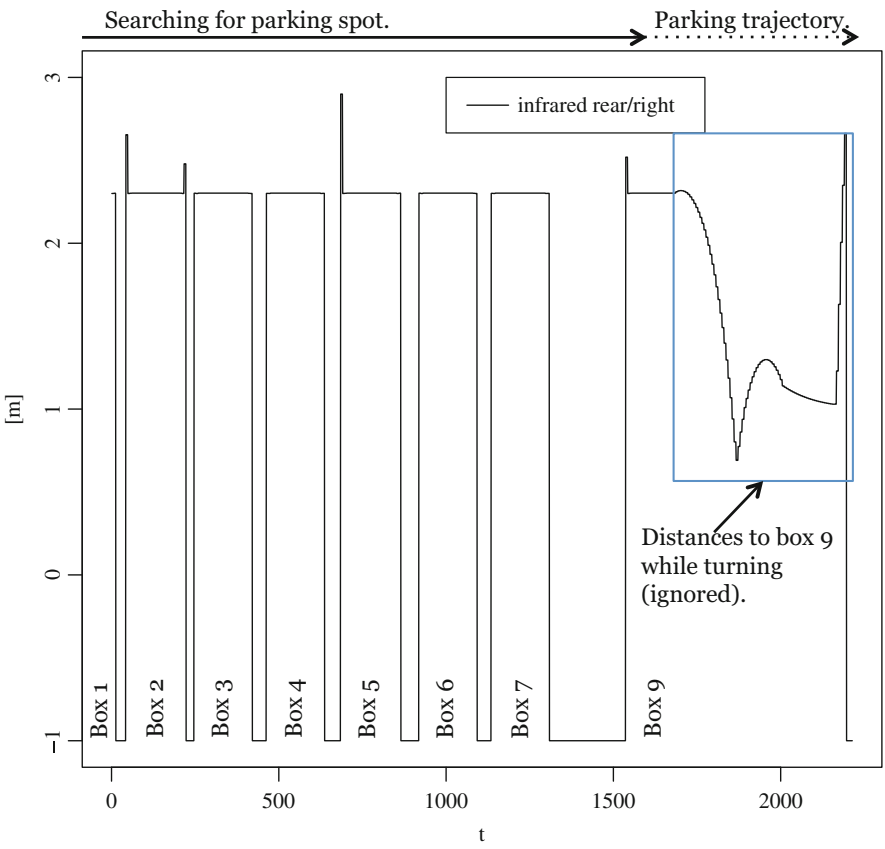


Fig. 10.3 Distances over time for the exemplary parking scenario for the distance sensor mounted at the rear/right corner during an experimental run in the simulation environment

an obstacle, the distance $d = -1$ is returned; otherwise, the measured distance is returned. The empty spot between box 7 and box 9 can be clearly identified.

10.2.3 State Machine for Box Parking

Based on the measurement profile depicted in Fig. 10.3, the concept for a state machine realizing a box-parking behavior can be derived. It is apparent that a parking gap is described by the following event sequence: $d > 0 \,!\, d < 0$ followed by $d < 0 \,!\, d > 0$. Between both events, the travelled distance needs to be observed to determine the size of the parking gap. If the size is wide enough, the parking trajectory can be initiated.

Figure 10.4 shows a state machine realizing the aforementioned event observer. Firstly, necessary variables are initialized before the state machine continues to the

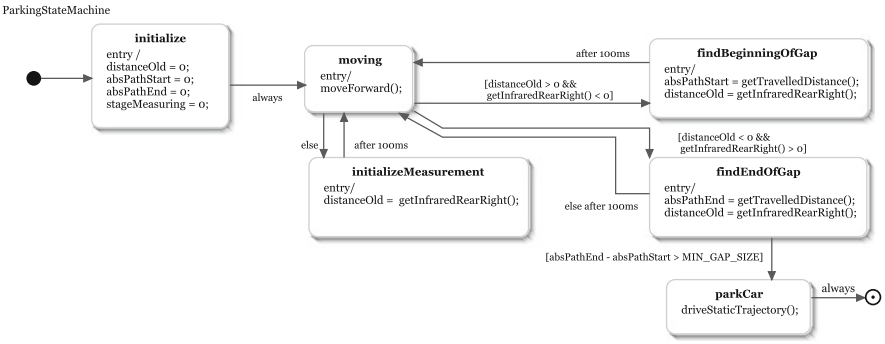


Fig. 10.4 Parking state machine with a static parking trajectory

moving state. In this state, the vehicle continues following the current lane. Next, the hysteresis variable distanceOld is set by the current value from the distance sensor.

In the next cycle, the currently measured value is compared to the previously mentioned event sequence to identify the first part. Once it has been found, the currently travelled path is saved in findBeginningOfGap. Subsequently, the second part of the event sequence is awaited to fire the transition to the state findEndOfGap. Once this state is activated, the currently travelled distance is compared to the previous value to eventually initiate the static parking trajectory in state parkGap if the found parking gap is wide enough.

10.3 Virtual Testing for the Box-Parking State Machine for Continuous Integration

While the automated parking algorithm can be integrated and tested in reality by using an experimental platform of a miniature car [8], such tests that depend on real hardware environments are time consuming and error prone. Furthermore, ensuring the repeatability of the conditions of the test environment to analytically investigate and understand potentially unexpected behavior is challenging.

As an intermediate step, the integrated software system can be systematically investigated in a virtual test environment. Such an environment replaces hardware components like sensors and actuators with virtual counterparts. These virtual components are used to stimulate the data processing chain of a CPS, for example, and to imitate physical interactions within the surroundings like driving according to a physical motion model.

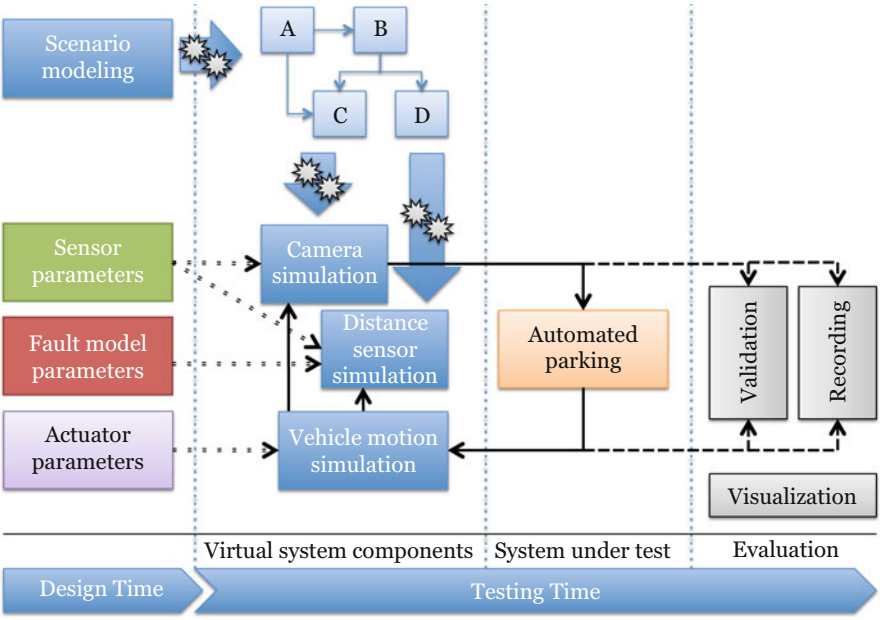


Fig. 10.5 Layers in a virtual test environment for simulation-based testing of a complex cyber-physical system

10.3.1 Design Considerations

Figure 10.5 depicts the general concept that was used to evaluate the self-parking algorithm. The concept consists of two phases, where the first one is the design time and the second one is the testing time.

In the former phase, the virtual test case is either manually defined or created from a model-based description of the scenario and situation that should be handled by the system. Furthermore, the parameters for the virtualized sensors of the CPS are defined like type of sensor, virtual mounting position in the 3D environment (e.g., on the car in the running example), pointing direction, and opening angle; for ray-based sensors, the viewing distance and resolution need to be modeled as well.

Next, the virtual actuators of the CPS need to be modeled; in the running example, this comprises the physical motion model of the vehicle and the interface to the higher software layers. As outlined in [8], platform-independent data structures in the processing chain in combination with a proxy layer/component that maps them to platform-specific representations enable the encapsulation of the concrete hardware interfaces for the virtual test environment.

We evaluated the box-parking algorithm in a simulation environment as originally described in [7]. This experimentation and development environment was adapted from a development environment that was already successfully used for the development of a real-scale self-driving car [9]; some of the concepts that are

realized in this experimentation and development environment were already developed and tested in the international competition 2007 DARPA Urban Challenge [5].

As depicted in Fig. 10.5, the core for this simulation environment is a domain-specific language (DSL) for the description of the stationary surroundings and dynamic parts of a situation. Instances of this DSL are used in a model-based virtual testing process to generate data structures for the virtual counterparts of the sensors, for example, as depicted by the dotted arrows; for the virtualized automotive use case, this includes a 3D representation in OpenGL to realize a monocular camera and ray-based distance sensors that are using the description of the surroundings to return measured distances to objects over time to imitate ultrasonic- and infrared-based distance sensors.

While these virtual counterparts are used to stimulate the data processing chain of the CPS, the virtual counterparts of the actuator layer of the real hardware environment are needed to close the feedback loop of the CPS. In the running example of a self-driving miniature car, the minimum turning radii in both turning directions as well as the wheel base are determined to create a vehicle motion model based on a linear bicycle system to simulate the vehicle movements. This feedback loop is shown in the center of Fig. 10.5 depicted by the solid arrows.

10.3.2 Automating Virtual Tests

To automate the evaluation of this feedback loop for many different scenarios, an additional layer shown as evaluation is needed. The components that are used in this layer do not interfere with the previously mentioned feedback loop. However, they also use the stimuli for the system under test (SUT) and its responses to validate if SUT behaves as expected; in the running example, the expected behavior is that the vehicle is following the lane on the right-hand side, identifies the first sufficiently wide parking gap, and parks backwards in the parking spot without touching any obstacles at any time.

This feedback loop in combination with the validation component, which returns true or false depending on the fulfillment of the expected behavior, would be enough to automate the simulation of the available scenarios in a continuous integration approach, for instance, in a cloud-based environment [10]. However, basing the defect localization and correction only on a Boolean result is not helpful when many different scenarios are used that test the same algorithm from various perspectives. Therefore, a recording component is required that additionally records all stimuli and response data. For a failing test in a continuous integration environment, replaying and visualizing the stimuli in a visualization tool while monitoring the state machine in the SUT facilitate the defect localization.

Such a layered architecture of the virtual test environment also enables the systematic and automated analysis of the system reliability and robustness of the CPS. In the running example, faults can be injected to the sensors, for example, to evaluate the system's behavior in cases of missing or unplausible sensor data.

Thus, the same set of scenarios that is used to test the expected behavior of the SUT can be reused with a model-based description of the faults to be injected.

Summary and Conclusion

In this article, the design of a virtual test environment to test cyber-physical systems is described. The specific challenge for such systems is their dependency on volatile data perceived from the surroundings by using sensors like cameras and radars; furthermore, these systems interact with the environment by using actuators. As a running example, automated box parking for a self-driving miniature car is described.

Testing such systems on real proving grounds or in field tests is time and Resource consuming. Furthermore, the repeatability of a specific scenario is already challenging on proving ground and hardly possible in random tests on public roads. Additionally, growing product families increase the need to address these challenges already at earlier test phases to make a more efficient use of the existing test resources.

In this regard, a virtual test environment can complement such real-world tests by replacing real hardware like sensors and actuators with virtual counter-parts. These counterparts enable a closed-loop simulation of various scenarios that stimulate the data processing chain of the system under test. To fully automate such simulations, validating components are needed that observe the behavior of the system under test over time to determine whether a given scenario is fulfilled. Furthermore, the layered architecture of such a test environment also enables systematic reliability and robustness analysis by reusing the set of existing scenarios while injecting faults from a model-based representation. In combination with a data logging interface and a visualization tool, the unexpected system behavior—even when it is identified during continuous integration in automated system tests—can be analyzed systematically later.

References

1. Hirsch, J.: Self-driving cars inch closer to mainstream availability. Los Angeles Times. URL http://www.latimes.com/business/autos/la-fi-adv-hy-self-driving-cars-20131013,0,5094627.story. Last accessed 14 Oct 2014
2. Ulmer, B.: VITA - an autonomous road vehicle (ARV) for collision avoidance in traffic. In: Proceedings of the Intelligent Vehicles '92 Symposium, pp. 36–41 (1992)
3. Thrun, S., Montemerlo, M., Dahlkamp, H., Stavens, D., Aron, A., Diebel, J., Fong, P., Gale, J., Halpenny, M., Homann, G., Lau, K., Oakley, C., Palatucci, M., Pratt, V., Stang, P., Strohband, S., Dupont, C., Jendrossek, L.E., Koelen, C., Markey, C., Rummel, C., van Niekerk, J., Jensen, E., Alessandrini, P., Bradski, G., Davies, B., Ettinger, S., Kaehler, A., Nefian, A., Mahoney, P.: Stanley: the robot that won the DARPA grand challenge. J. Field Robot. **23**(9), 661–692 (2006)
4. Montemerlo, M., Becker, J., Bhat, S., Dahlkamp, H., Dolgov, D., Ettinger, S., Haehnel, D., Hilden, T., Homann, G., Huhnke, B., Johnston, D., Klumpp, S., Langer, D., Levandowski, A., Levinson, J., Marcil, J., Orenstein, D., Paefgen, J., Penny, I., Petrovskaya, A., Pflueger, M.,

Stanek, G., Stavens, D., Vogt, A., Thrun, S., Artificial, S., Cs, S., Hähnel, D.: Junior: the Stanford entry in the urban challenge. In: The DARPA Urban Challenge. Number October 2005, pp. 91–123 (2009)
5. Rauskolb, F.W., Berger, K., Lipski, C., Magnor, M., Cornelsen, K., Eertz, J., Form, T., Graefe, F., Ohl, S., Schumacher, W., Wille, J.M., Hecker, P., Nothdurft, T., Doering, M., Homeier, K., Morgenroth, J., Wolf, L., Basarke, C., Berger, C., Gülke, T., Klose, F., Rumpe, B.: Caroline: an autonomously driving vehicle for urban environments. J. Field Robot. **25**(9), 674–724 (2008)
6. Augusto, B., Ebadighajari, A., Englund, C., Hakeem, U., Irukulapati, N.V., Nilsson, J., Raza, A., Sadeghitabar, R.: Technical aspects on team Chalmers solution to cooperative driving. Technical report, Chalmers University of Technology, Göteborg (2011)
7. Berger, C., Chaudron, M., Heldal, R., Landsiedel, O., Schiller, E.M.: Model-based, composable simulation for the development of autonomous miniature vehicles. In: Proceedings of the SCS/IEEE Symposium on Theory of Modeling and Simulation, San Diego, CA, USA (Apr 2013)
8. Berger, C.: From a competition for self-driving miniature cars to a standardized experimental platform: concept, models, architecture, and evaluation. J Softw. Eng. Robot. **5**(1), 63–79 (2014)
9. Berger, C.: Automating acceptance tests for sensor- and actuator-based systems on the example of autonomous vehicles. In: Aachener Informatik-Berichte. Software Engineering, Band 6. Shaker Verlag, Aachen, Germany (2010)
10. Berger, C.: Cloud-based testing for context-aware cyber-physical systems. In: Tilley, S., Parveen, T. (eds.) Software Testing in the Cloud: Perspectives on an Emerging Discipline, pp. 68–95. IGI Global, Hershey (2012)

Chapter 11
Industrial Application of Visual GUI Testing: Lessons Learned

Emil Alégroth and Robert Feldt

Abstract A large body of academic knowledge has been devoted to automated software testing in order to support the software market's demands for continuous software delivery. However, most of these automated techniques approach testing from lower levels of system abstraction, e.g., component level, which limit their applicability for high-level regression testing of, for instance, system and acceptance tests, thus forcing companies to perform these test activities manually, which is considered time consuming, tedious, and error prone.

In this book chapter, we present visual GUI testing (VGT), a tool driven test technique that uses image recognition in order to interact and assert the correctness of a system under test (SUT) through the bitmap graphical user interface (GUI) that is shown to the user on the computer monitor. This approach makes VGT flexible and applicable to any SUT with a GUI but also allows VGT tools to emulate end-user behavior and therefore automate previously manual system and acceptance test cases. In addition to presenting the technique itself, this chapter will also present some VGT tools and empirically identified problems with the technique and how these problems can be mitigated in practice. Finally we will discuss how VGT can be used in the context of continuous software development in order to support market demands for quicker software delivery.

E. Alégroth (✉)
Software Engineering and Technology, Chalmers University, Gothenburg, Sweden
e-mail: Emil.Alegroth@Chalmers.se

R. Feldt
Software Engineering and Technology, Chalmers University, Gothenburg, Sweden

Department of Software Engineering, Blekinge University of Technology, Karlskrona, Sweden
e-mail: Robert.Feldt@bth.se

J. Bosch (ed.), *Continuous Software Engineering*,
DOI 10.1007/978-3-319-11283-1_11

11.1 Introduction and Background

The time available between releases of software in industry is becoming shorter due to growing market demands for continuous software delivery. This trend is both facilitated and spurred on by agile and lean development processes, e.g., Scrum and Kanban, and in extension the practices of continuous integration and deployment [1]. However, the faster development pace can also have detrimental effects on software quality, e.g., since less time is available for software testing.

To mitigate such detrimental effects, a large body of research has been devoted to test automation techniques and tools. A majority of this research has focused on test automation on lower levels of system abstraction, e.g., unit testing [2]. Lower-level testing is powerful for verification of software components, but for higher levels such as system testing, these techniques become cumbersome, complex, and costly [3, 4]. Automated techniques for high-level testing exist but are infrequently used in practice due to a lack of or immature tooling and because companies have faulty expectations on the test automation, e.g., that automation will lower the cost of testing [3]. Therefore, common industrial practice is still to use mainly manual system test practices, often performed through the system under test (SUT)'s graphical user interface (GUI) following predefined system usage scenarios [3]. However, scenario-based manual testing is time consuming and is therefore not a practical option to facilitate the market's need for faster software delivery [3].

Visual GUI testing (VGT) is a novel[1] automated test technique that helps fill the gap for automated GUI-based system testing [5–8]. The technique is tool driven and uses image recognition in order to identify and interact with a SUT's GUI as it is represented graphically to the end user. As such, VGT can be used to emulate end-user behavior and automate tests that previously had to be performed manually. Consequently it can lower test execution costs which facilitates higher test frequency and thus leads to more and more frequent feedback on the SUT's quality to developers. Our previous research has shown that VGT is applicable in industrial practice, but only limited information exist about the technique's long-term feasibility [5–8]. As such, it is unclear if VGT is a suitable technique to improve companies' continuous integration and deployment practices.

11.2 Software Testing

Software testing refers to the practice of verifying or validating an SUT's conformance to its requirements. We define a system as verified if it operates according to its requirements.[2] We define a system as validated if it operates according to its

[1] Even though the general idea has been around for long, it has only been made possible through recent advances in computer power.

[2] An additional complication is that not all requirements might have been (correctly) elicited, i.e., might not properly represent the actual needs of the users; however, in the following, we do not go further into this distinction.

requirements and as intended by the customer. Hence, there is an important distinction between verification and validation that also separates scenario-based system tests from acceptance tests. While system tests can be any scenario that verifies the systems functionality, an acceptance test must follow a scenario of how a user will actually use the system. Scenario-based testing is as such one of the primary ways of ensuring software quality and can also be performed automatically through GUI-based testing [9].

11.2.1 Manual Testing

Still, manual practices are the most common for system and acceptance testing in industry, either with guiding scenarios or using exploratory approaches [10]. While exploratory tests are aimed at identifying previously unknown defects, scenario-based manual tests are aimed at verifying continued conformance to the system's requirements, i.e., regression tests. Regression tests thereby aim to verify that changes to the SUT have not broken functionality that was previously functional.

Manual test scenarios generally define user interaction with the SUT or its environment through specification of inputs and expected outputs to said inputs, thus limiting the defect finding ability of these tests to defects that are explicitly asserted by the scenario. This limitation imposes a need for scenario-based testing to be complemented with exploratory tests, regardless if the scenario-based tests are manual or automated.

In addition, manual testing is slow, resource intensive, and, in the context of continuous integration and deployment, a potential bottleneck, hence enforcing the need for test automation to remove or mitigate the need for manual tests.

11.2.2 Automated Testing

There is a plethora of automated test techniques and tools for different testing purposes [11], but we will only briefly cover a few techniques in this section for the continued discussion. In contrast to manual testing, the execution of automated tests are not associated with any large cost, which facilitates frequent feedback to the developers. The main benefit of frequent feedback is that defects are found quicker and their root cause sought in a smaller part of the software code base. In this way, there is less accumulation of defects and a lesser chance of synergistic effects, e.g., failure masking, that can result in higher overall development cost. To explain, assume that manual tests are run once every three weeks of development. During the development, it is possible that several defects are introduced in the SUT and that there are dependencies among the said defects. These dependencies can make defect analysis and identification problematic because the source of one defect can obscure the source of another defect. With continuous automated testing, it is

possible to find the individual defects quicker, and the synergy effects can be averted. This can lower the overall cost spent on testing and corrective actions.

However, there are other costs associated with automated testing, i.e., development and maintenance costs. These costs are affected by several different factors that differ from the development and execution of manual test cases, such as a need/want for tester technical knowledge and experience, test and tool complexity, etc. Furthermore, the humans can use their cognitive ability to determine how to perform a test case. As such, a manual test case can be written on a higher level of abstraction and be less formal than an automated test, i.e., the automated test requires the exact instructions that it should execute in order to test the SUT. Therefore, automated tests are prone to need more continuous maintenance than manual test cases.

However, keeping automated tests up to date is imperative to ensure SUT quality but also for the automated test technique in question's longevity. To explain, test suites that are not continuously maintained quickly degrade, and a critical point is quickly reached when the cost of maintenance outweighs the benefits of the tests [3]. As such, companies that end up in this scenario often revert back to previous test practices, e.g., manual testing.

As previously discussed, we separate automated testing into levels of system abstraction. Low-level tests are generally white box that means that they require access to the source code or information about the SUT's implementation. An example of such a technique is unit testing with xUnit [2]. These tests are dependent on the programming language and operate by asserting code components with inputs and expected outputs. For instance, a method that increments a variable by one could be tested by setting a variable to a value x, e.g., three, and then verified against the expected value $x+1$, i.e., four in this case. Such a test could be considered simplistic, but given a large test suite with assertions with a wide range of input and output pairs, it has been shown to be a powerful tool in finding defective system behavior. However, it has also been shown that these tests are ill suited for higher-level tests, e.g., system tests [3]. The reason is because system tests written with XUnit quickly become complex because the tests need to consider both the logical and chronological dependencies between the software components of the SUT.

Another approach to automated testing is based around test case generation, e.g., with model-based testing. Theoretically, test case generation can be applied for all abstraction levels of testing, but focus in research has primarily been on lower-level tests. One plausible reason for this focus is because higher-level tests become more intricate and require more knowledge about the system, which makes it difficult to create the test generation and oracle algorithms. The work of Memon et al. does however show that test generation of GUI-based tests is possible [12, 13].

High-level tests, on the contrary to low-level tests, test the system's components as a whole and how they work together. As such, these tests require less knowledge about a system's inner workings and are therefore often gray box or even black box. Gray-box tests are tests that require limited access to source code or properties of the source components, e.g., properties from the GUI accessed through the SUT's

GUI libraries. Black-box tests in contrast require no access and can test a component or system without knowing any of its inner workings.

11.2.3 GUI-Based Testing and the First Generation

Graphical user interface (GUI)-based testing refers to system-level testing through a system's GUI. Hence, it does not refer to testing of the GUI but rather the system's functionality through its GUI. This distinction is important but generally misconceived in practice. As we have previously discussed in Sect. 11.2.2, testing can be performed on different levels of system abstraction. GUI-based tests are performed on a high level of abstraction and therefore allow the user to automate more intricate scenarios that stimulate a larger subset of the SUT's components. Thus, even short test cases on high level of abstraction can be equivalent to many low-level tests. However, at the same time, this property dislodges the tests from the source code. To explain, while low-level tests, e.g., XUnit, can tell the user what exact component or even line of code is defective, higher-level tests can only present what feature is defective in the system. As such, the results of these tests require the tester to analyze the SUT in order to find the source code responsible for the defect.

There are several techniques for GUI-based testing, which we refer to as generations of GUI-based testing. The first generation, also known as coordinate-based GUI-based testing, used exact coordinates on the screen to interact with the SUT's GUI. This approach was beneficial for test script recording, but because GUIs are prone to change, these tests also required a lot of maintenance. Because of the high maintenance costs, this technique has been abandoned and replaced with what we refer to as the second-generation GUI-based testing.

11.2.4 Second-generation GUI-Based Testing

Property or widget-based GUI-based testing is a commercially used technique that can be used for system-level testing [3]. The technique is gray box and uses the properties of GUI components in order to identify and interact with them. To explain, consider the GUI as shown to the user as a representation of the GUI code components on a lower layer in the systems architecture. The code defines information about the GUI components characteristics, such as size, color, labels, etc., information that second-generation GUI-based testing tools use in order to uniquely identify specific components and assert their state.

The benefits of this technique are that it can be used for test case recording but also that it is robust to GUI layout change. Hence, both the GUI graphics and the location of GUI objects can be changed without any need for maintenance of the test scripts. However, since the scripts use underlying information of the GUI

components, changes to the source code can cause the test scripts to fail. Further the technique is limited to a set of programming languages, dependent on which languages the tool in question supports. In addition, some types of GUIs cannot be tested, for instance, dynamic GUIs where output is drawn on, for instance, a canvas in runtime, thus limiting the technique's applicability.

A common usage of this technique is for automated web testing with the tool Selenium. Another commonly used tool is HP's Quality Test Professional (QTP) or, as it is now known, Unified Functional Testing (UFT).

11.2.5 Visual GUI Testing (VGT), the Third Generation

Visual GUI testing is an emerging technique in industrial practice that uses image recognition in order to identify and interact with the GUI through the bitmap graphics shown to the user during system runtime [5–8]. Interaction is then performed using the operating systems mouse and keyboard functionality, emulating user interaction with the SUT. The use of image recognition also makes the technique flexible and allows the technique to be used on any GUI-based system regardless of implementation or even platform. As such, the technique is black box and does not suffer from the applicability limitations presented for second-generation testing. As an example, assume that we have an application that given a dataset draws a graph on a canvas. If the said graph is deterministic, a VGT tool can assert the SUT if the graph is drawn correctly or not, an assertion that was previously not possible with other GUI-based tools.

VGT is a tool-driven technique, and several commercial and open-source tools are currently available for practice. However, even though there are several tools available, the use of the technique in industrial practice is low. Some reasons are because the technique is still immature, because few companies know about it still, and because there is little to no support about the feasibility of the technique for long term. For instance, only limited information has been acquired regarding the maintenance costs associated with VGT scripts. However, the information that does exist is indicative of VGT being a suitable complement to other automated and manual test practices in practice.

We state that VGT is a complement, not a replacement, to other techniques because VGT still can only find defects in the assertions that are added to the VGT scripts. To explain, assume that we have a defective calculator application that when you click on the number 3 button the number 9 button is also highlighted that it is being clicked. Thus, if we assume that we have a test case that asserts that when the number 3 button is pressed it changes state, such a test case would not capture the defective number 9 button and as such cause fault slip through. Hence, the VGT scripts need to be complemented with explorative manual testing that can uncover

defects that lie outside the scripted scenarios. VGT is therefore primarily a regression testing technique, even though proof-of-concept research shows that it can be used also for random testing to find new defects.

11.3 VGT Tools

The first academically reported VGT tool was called Triggers, published in a paper by Potter in the early 1990s [14]. However, because of performance issues, the tool and the technique was not considered a feasible approach in practice.

Since then, advances in hardware and image recognition algorithms have resulted in a renewed interest in the technique, and several tools are currently available in the market. In the following section, we present two VGT tools and their core features.

11.3.1 Sikuli

Sikuli[3] is an open-source VGT tool developed originally by researchers at the Massachusetts Institute of Technology (MIT) and was presented by Yeh et al. in 2009 [15]. The tool uses Python as its primary scripting language but with an extended application programming interface (API) that makes use of the tool's image recognition capabilities. As such, the entire Python language is at the disposal of the user in the tool's integrated development environment (IDE). Furthermore, since the Sikuli API methods take images as input, the tool's IDE implementation displays the said images in the script files, thus making scripts more readable and making it easier to perform VGT. Figure 11.1 shows a screenshot of what the IDE with a short Sikuli script looks like. The tool also has a JAVA API, but at the time of writing this book, it was not as developed as the Python API.

However, the tool's use of these script/programming languages provides the user with support to write more advanced test applications, e.g., random and explorative testing. The tool also supports optical character recognition, which is valuable for testing of applications with non-deterministic textual output and for the creation of reusable data-driven scripts, i.e., scripts that can be executed with different input and expected output taken from a source file or database.

However, the tool is quite immature and associated with some quality concerns in regard to robustness [8]. Furthermore, the tool does not support script recording and only has limited support for test case management. The latter limitation originates in Sikuli primarily being an automation tool rather than a testing tool.

[3] More information available at www.sikuli.org.

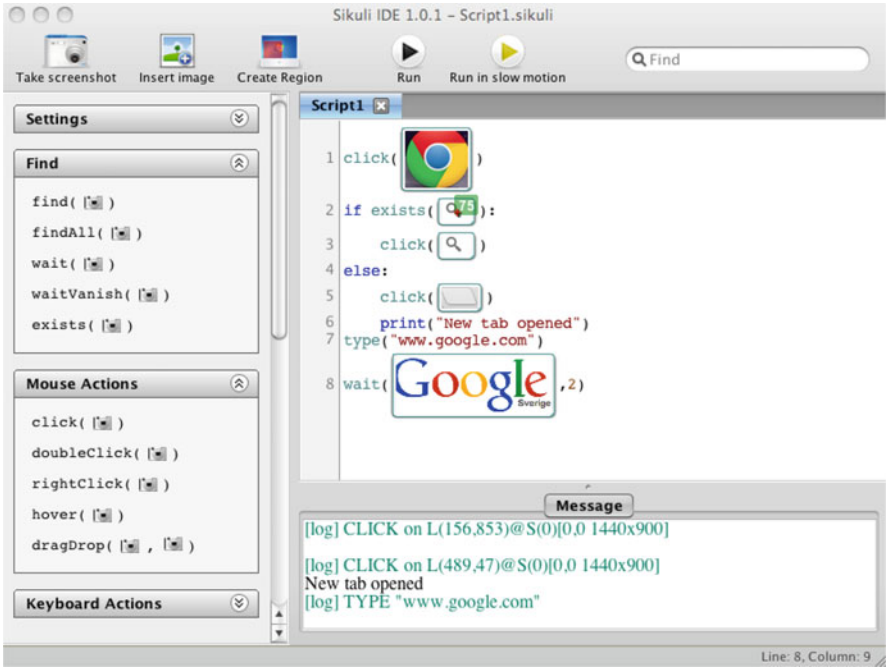


Fig. 11.1 Screenshot of Sikuli's IDE after a small script has been executed that opens a web browser and navigates to a search engine

However, the lacking support for testing can easily be developed because of the tool's available APIs.

11.3.2 JAutomate

JAutomate[4] is a commercial tool developed by the company Swifting in Gothenburg [7]. The tool supports both recording and manual development of scripts through a custom scripting language designed to be as easy to use as possible for novice programmers. Furthermore, the scripts can be viewed in different modes, with different levels of detail, to suit also more experienced script developers. Figure 11.2 shows a screenshot of the tool's IDE.

The tool also supports the inclusion of manual test steps in a test script and includes several image recognition algorithms. The most basic algorithm only checks that the pixels in the sought image are the same as the found image, using a random search pattern. A second algorithm does post-capture work of the sought

[4] More information available at www.jautomate.com.

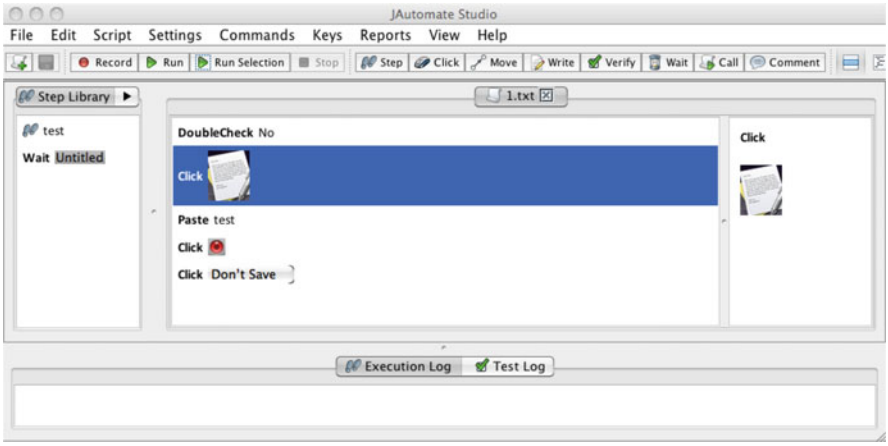


Fig. 11.2 Screenshot of JAutomate's IDE

image and extrapolates the areas in the image with the highest contrast. During script playback, the algorithm searches for the extrapolated areas which have been shown to improve execution speed. Finally the tool has a more robust algorithm that uses more information in the sought image but also its surrounding bitmaps, i.e., context information where the sought image is located in the GUI, to find a match. This algorithm is useful when searching for objects with non-deterministic output given that the surrounding bitmaps remain deterministic.

In contrast to Sikuli, JAutomate was originally designed for testing and therefore has support for both test suite creation and management. Additionally, the tool includes support to be executable from a build server, i.e., it can be integrated into an automated software build environment that is required for continuous integration and deployment.

11.4 Development of VGT Scripts

There are different approaches to VGT script development. The most common, as reported by research, is to translate manual test cases into automated scripts. However, research also reports that there are several aspects to the development that must be considered to make VGT as effective and cost-efficient as possible. In the following section, we will briefly discuss some of these aspects.

11.4.1 Development Guidelines

Test automation has several similarities to traditional software development, and therefore, it is suitable to use similar practices when developing test code, for instance, to use a coding standard, properly design the test framework/architecture, etc.

However, there are also dissimilarities that must be considered. For instance, VGT scripts need to be synchronized both logically and chronologically against the state transitions of the SUT being tested. In order to do synchronization effectively, it has been observed that the script developer needs to have a linear and systematic mindset and sufficient knowledge of the purpose of the test case, the SUT's features, and the VGT tool's capabilities. In addition, script linearity is important to lower script complexity, i.e., it has been observed that scripts with loops and branches quickly become more complex. Keeping the script complexity as low as possible is paramount for script readability and in extension to mitigate maintenance costs. Consequently, manual test cases should only be used as specification for the VGT scripts if they define linear test scenarios.

Another favorable guideline to mitigate maintenance cost is to start by writing tests for the SUT's most stable features, i.e., test cases for features that are not likely to change. Additionally, it is recommended that the implementation is performed incrementally rather than big bang, especially in contexts where it is uncertain if VGT is a suitable improvement to the development process.

11.4.2 Test Architecture and Design

In order to get the most benefit from VGT, it is important to consider the test architecture and design it to be as modular as possible. Modularity improves modifiability and reusability of test components that in turn improves the tests flexibility and ability to respond to change, consequently mitigating both development and maintenance costs. In addition, this design allows for the creation of reusable support scripts, e.g., to start a simulator, which are required to run a test scenario, but is not explicitly part of the said scenario.

For individual scripts, it is also suitable to define a guideline for how to create the scripts in order to ensure consistency. Consistency relates both to coding standards, i.e., the look and feel of the actual test code, but also how components within a test script are structured. For instance, following the setup, test and teardown structure commonly used by XUnit frameworks. Keeping the scripts consistent will improve readability and therefore mitigate maintenance costs.

In summary, it is important to plan and structure the VGT test architecture prior to development in order to maximize its use and minimize cost. Furthermore, in a context where continuous integration and/or deployment is requested, it is also important to consider how the architecture can support such a practice.

11.5 Challenges with Applying VGT

VGT is still, despite its benefits, an immature technique, and its practical application is associated with several challenges, problems, and limitations (CPLs). In this section we will discuss some of the core CPLs that have been collected through empirical work in industrial practice [8].

11.5.1 Expectations

The first core CPL is not associated with just VGT but automated testing in general. A common misconception with automated testing is that it will lower the cost/time spent on testing. This is however generally false because even though automated tests are not associated with any execution cost, they still require maintenance. Maintenance that can be equal to or even greater than the costs of running the tests manually. However, in order to evaluate the value of automated testing, one also needs to consider the increased number of executions and resulting quality gains, a statement supported by empirical work where industrial practitioners state that the value of automation outweighs the maintenance costs.

11.5.2 Synchronization

The second core CPL is not general to all automated testing but general to all GUI-based test techniques and refers to the synchronization between test script and SUT state transition. In the case of VGT, this means that the tools need to use graphical output from the SUT's GUI to verify that the SUT is ready to receive input. To explain the problem, consider a webpage with several pages. If a link on the page is clicked, it can take several seconds before the next page is loaded (depending on network latency). A human will automatically wait for the webpage to load but the scripts need to be told to do so. Therefore, all VGT tools include methods to wait for images on the screen in addition to static waits.

However, verifying that a script executes correctly can only be done by viewing it during execution, and for long test scenarios, this practice can become quite tedious, especially if the script fails in its final stages due to incorrect timing. This problem enforces the guideline that VGT scripts should be kept linear but also as short as possible to lower the required time to verify script correctness.

11.5.3 Image Recognition

The third core problem is VGT specific and relates to the image recognition of the currently available tools. It has been empirically observed that VGT tools sometimes without reason fail to find an image, thus leading to false-positive test results. The only strategy that has been found to mitigate this problem is to add redundancies in the script code such that if the image recognition fails, it is automatically rerun or executed with another sought image. However, as the tools of the technique mature, this problem is perceived to dissipate.

11.5.4 Others

VGT tools also suffer from other immaturity problems, e.g., lack of functionality or instability. These problems have been encountered in several empirical studies but have been reported as manageable either with script development practices, e.g., how screenshots are taken and coding standards, or technical solutions, e.g., adding layers of failure mitigation in the script architecture. However, no detailed guidelines or practices have been identified that are applicable in all contexts for all systems.

Furthermore, because of the immaturity, the amount of support available to script developers online is limited, e.g., communities focused on VGT. However, as the tools and usage of VGT improves, this is considered to become less of a problem.

Another empirically found problem is that VGT scripts based on manual test specifications are also reliant on the said specifications being correct and up to date, i.e., that they are properly aligned and up to date with the system under test. This problem is general to all testing and is caused by improper documentation management during the evolution of a software system.

11.6 VGT for Continuous Integration

VGT is perceived to be applicable to support continuous integration and deployment, but no research results exist to support this statement.

Automated testing is an essential part of continuous integration and deployment to quickly ensure software quality before release. As such, the execution speed is one of the most important properties for continuous automated testing, i.e., the tests should preferably be executed every build. This property is however a potential problem for VGT scripts that cannot execute faster than the system can respond, as discussed in Sect. 11.5.2. As such, a VGT suite generally has an execution time in the order of hours, compared to a unit test suite with an execution time in the order

of seconds. Consequently, it might not be feasible to run the VGT suite every build but rather only once a day. Thus, VGT would not be applicable for hourly continuous integration and/or deployment. However, further research is required to evaluate this statement.

Another potential problem in a continuous testing context is the amount of maintenance that would be required to keep the VGT suite up to date. Since continuous integration and deployment infer that the software changes frequently, it is further important to prioritize what features the tests are developed for first, as discussed in Sect. 11.4.

The benefit of using VGT in a continuous integration build chain is perceivably that test results from the unit testing can be combined with the VGT test results to find not just what feature is defective but what component is the source of the said defect, thus shortening the defect analysis and identification time by providing the developer with test feedback on several levels of system abstraction simultaneously. In addition, the GUI level feedback itself is valuable for continuous integration and deployment and perceivably mitigates the need for manual testing, i.e., less manual testing is required.

In summary, VGT is perceived to be applicable for continuous integration and deployment but only if integration is done on a daily basis and assuming that the maintenance costs can be kept feasible. However, further research is required to verify these statements and to find support for VGT's use in a continuous integration and deployment context.

Summary and Conclusions

Visual GUI testing (VGT) is a technique for automated system and acceptance testing that has recently been made practical for industrial application. It uses image recognition to interact with the graphical user interface of a software system and can thus emulate actual use of the system, regardless of its technical implementation. Our research has shown that VGT can be a valuable addition to current automated test techniques, especially for continuous integration and deployment, and we provide detailed guidelines and support for practitioners that want to apply it. However, currently no research has explored the benefits of using VGT for continuous testing, which is therefore a subject of future research.

References

1. Olsson, H.H., Alahyari, H., Bosch, J.: Climbing the "stairway to heaven" – a multiple-case study exploring barriers in the transition from agile development towards continuous deployment of software. In: 2012 38th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), pp. 392–399. EUROMICRO, IEEE (2012)
2. Olan, M.: Unit testing: test early, test often. J. Comput. Sci. Coll. **19**(2), 319–328 (2003)

3. Berner, S., Weber, R., Keller, R.: Observations and lessons learned from automated testing. In: Proceedings of the 27th International Conference on Software Engineering, pp. 571–579. ICSE, ACM (2005)
4. Grechanik, M., Xie, Q., Fu, C.: Creating GUI testing tools using accessibility technologies. In: International Conference on Software Testing, Verification and Validation Workshops, 2009. ICSTW'09, pp. 243–250. ICST, IEEE (2009)
5. Börjesson, E., Feldt, R.: Automated system testing using visual GUI testing tools: A comparative study in industry. In: 2012 I.E. Fifth International Conference on Software Testing, Verification and Validation (ICST), pp. 350–359. ICST, IEEE (2012)
6. Alegroth, E., Feldt, R., Olsson, H.: Transitioning manual system test suites to automated testing: An industrial case study. In: 2013 I.E. Sixth International Conference on Software Testing, Verification and Validation (ICST), pp. 56–65. ICST, IEEE (2013)
7. Alegroth, E., Nass, M., Olsson, H.: JAutomate: A tool for system-and acceptance-test automation. In: 2013 I.E. Sixth International Conference on Software Testing, Verification and Validation (ICST), pp. 439–446. ICST, IEEE (2013)
8. Alégroth, E., Feldt, R., Ryrholm, L.: Visual GUI testing in practice: challenges, problems and limitations. Empir. Softw. Eng. 1–51. Springer US (2014)
9. Humble, J., Farley, D.: Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation. Book, Pearson Education (2010)
10. Itkonen, J., Rautiainen, K.: Exploratory testing: A multiple case study. In: 2005 International Symposium on Empirical Software Engineering, 10 pp. (Nov 2005)
11. Dustin, E., Rashka, J., Paul, J.: Automated Software Testing: Introduction, Management, and Performance. Book, Addison-Wesley Professional (1999)
12. Memon, A.M., Pollack, M.E., Soffa, M.L.: Automated test oracles for GUIs. In: ACM SIGSOFT Software Engineering Notes, vol. 25, pp. 30–39. ACM (2000)
13. Memon, A.: An event-flow model of GUI-based applications for testing. Softw. Test. Verif. Rel. **17**(3), 137–157 (2007)
14. Potter, R.: In: Triggers: GUIding Automation with Pixels to Achieve Data Access, pp. 361–382. Center for automation research, University of Maryland (1992)
15. Yeh, T., Chang, T., Miller, R.: Sikuli: Using GUI screenshots for search and automation. In: Proceedings of the 22nd Annual ACM Symposium on User Interface Software and Technology, pp. 183–192. ACM symposium of UI software and technology, ACM (2009)

Part IV
R&D as an Innovation System

This part discusses the final step on the Stairway to Heaven, i.e., the use of continuous deployment to build a capability for R&D to continuously validate the development of new features by collecting data on the use of these features by customers. The fourth step in the stairway to heaven, continuous deployment, is not present explicitly in this book as it often evolves automatically as a natural extension of continuous integration. There are two chapters in this part. The first chapter presents the results of case studies at several Software Center companies concerning the current state of customer data collection. One of the key findings is that these companies today already collect terabytes of data but are unable to answer even the most basic questions concerning feature usage. The second chapter introduces the Hypothesis-Experiment (HYPEX) model. The HYPEX model explicitly outlines how to organize R&D in much faster iterations where thin slices of a feature are built and tested with customers. The HYPEX model allows companies to validate the value of building a feature by constantly measuring, as the feature evolves, how customers are using the feature.

Chapter 12
Post-deployment Data Collection in Software-Intensive Embedded Products

Helena Holmström Olsson and Jan Bosch

Abstract Companies collect terabytes of data about their products in the field, but research shows that R&D makes little use of this data, i.e., it is an untapped resource. In this paper, we explore collection and usage of post-deployment product data. We highlight the existing limitations in post-deployment data usage and the untapped resource that post-deployment product data remains. Based on a multiple case study at three software development companies, we collected three main findings. First, post-deployment data is used as input to the next pre-development phase, but not for improvement of existing product versions. Second, post-deployment data is used for troubleshooting and support activities, but not for innovation of new features. Third, post-deployment data provides a system-level understanding of operation and performance, but does not provide insight in individual feature usage. Finally, we propose a framework in which we outline what development practices and organizational mechanisms that need to be in place for advancing the usage of post-deployment data and advance the development of software-intensive embedded products.

12.1 Introduction

Today, products within telecommunication, transportation, consumer electronics, home automation, security, etc., involve an increasing amount of software. As a result, organizations that have a tradition within hardware development are transforming to become software-intensive organizations with software being responsible for a majority of the functionality, as well as for a majority of the development costs and investments. In this transition, the ability to learn about

H.H. Olsson (✉)
Department of Computer Science, Malmö University, Malmö, Sweden
e-mail: helena.holmstrom.olsson@mah.se

J. Bosch
Department of Computer Science and Engineering, Chalmers University of Technology, Gothenburg, Sweden
e-mail: Jan@JanBosch.com

J. Bosch (ed.), *Continuous Software Engineering*,
DOI 10.1007/978-3-319-11283-1_12

customers, and especially the way in which customers use software functionality, becomes increasingly important. Hence, agile software development practices that are flexible, responsive, and adaptive to customers [1, 2] are gaining momentum. In advocating customer collaboration and the importance of test-driven development practices [2], agile practices have attracted not only small software development companies but also companies involved in large-scale development of software-intensive embedded products.

However, while many companies have succeeded in applying agile practices and, as a result, leveraged the benefits of close customer collaboration and continuous validation of functionality in pre-deployment phases, there are few examples of companies that have succeeded in maintaining this close relationship to customers also after product deployment. One technique that has emerged due to the online nature of most software-intensive products is the opportunity to collect post-deployment data, i.e., data generated by the product after commercial deployment. This data can be operational data reflecting product performance, it can be diagnostic data recording product behavior, and it can be data indicating feature usage. For online technologies such as Web 2.0 software and software-as-a-service (SaaS) systems, the collection of post-deployment data is a well-established technique. In this domain, companies like Microsoft [3] and Intuit [4] successfully collect post-deployment data and use this as a basis for continuous improvement of existing products, as well as for input to innovation and new product development.

Interestingly, the opportunity to collect post-deployment data extends also to software-intensive embedded products [5]. Today, these products are increasingly connected, bringing with it the opportunity to collect data from real-time usage. For example, companies developing products within the telecom and automotive industry, i.e., mobile phones and cars, are starting to explore the advantages of collecting data from their products in the field.

In this chapter, we present a multiple case study on three companies developing software-intensive embedded products. While in different domains, all companies develop products consisting of an increasing amount of software, and they all collect large amounts of data from the products they release. In our study, we explore the following research questions:

- *What* post-deployment data do the companies involved in our study collect?
- For *what purposes* is this data used?

The contribution of the paper is twofold. First, we identify what data companies collect and the current limitations in data usage. We also outline the key opportunities that improved data usage would bring with it. Second, we propose a framework for organizations interested in advancing their usage of post-deployment data. Our framework reflects the different levels of post-deployment data usage, as well as the mechanisms needed for improving post-deployment data usage.

12.2 Background

12.2.1 Agile Software Development

During the last decade, agile development methods have dramatically changed the way software development is performed. Agile methods are characterized by short development cycles, close customer collaboration, rapid feedback loops, and continuous evaluation of functionality [2, 6]. In comparison to plan-driven development methods, agile methods operate on the principle of "just enough method" and seek to avoid processes that add little value to the customer. While the agile principles were initially developed for smaller software development organizations, evidence show that large software-intensive organizations operating in complex global development environments are in the process of deploying agile methods as part of their de facto approaches to software development.

Typically, agile methods focus on collecting customer feedback during pre-deployment phases, i.e., before and during development. Techniques such as use cases, scenarios, prototyping, stakeholder interviews, joint requirements sessions, joint application design sessions, etc., are common. Likewise, techniques such as alpha and beta testing, observation, expert reviews, and prototyping are efficiently used during development in order to continuously validate that the functionality that is developed is of value to the customers. As can be seen in previous research [2, 7–9], these techniques are successfully used to capture generic customer needs for mass-market products [10]. In addition, large-scale development companies often use product management as a proxy for communicating customer feedback to the development organization before and during development of the system [11].

However, while agile development practices are conducive to close customer collaboration and continuous validation of functionality in the early phases of development [6, 12], there is less evidence on companies that have succeeded in establishing techniques for continuously collecting customer feedback also after commercial deployment of the product.

12.2.2 Post-deployment Customer Feedback

As product use evolves over time, product characteristics need to be adjusted, adapted, and updated according to emerging customer behaviors and needs. This implies that mechanisms for post-deployment customer feedback are as important as those used during the pre-development and development phases of a product.

Recently, and with the increasing number of software products being Internet connected, new opportunities for observing and measuring post-deployment product behavior and use have emerged [3, 4]. The most well-known examples of this are Web 2.0 technologies, social network systems, and software-as-a-service (SaaS)

systems. Due to the online nature of these systems, data is generated and can be collected as soon as customers use the systems, and the cost of collecting data from, and about, the customer is low [4, 5]. Examples include the amount of time a user spends using a feature, the frequency of feature selection, the path that a customer takes through product functionality, etc. If continuously collected and analyzed, product data can be used as efficient input for improvement of the existing product and as a basis for new product development and innovation. As a result, these online systems allow for an approach where instead of freezing the requirements before development starts, requirements evolve in real time based on data collected from the products.

Interestingly, and as the focus of this paper, these benefits extend also to software-intensive embedded products. Today, companies developing connected embedded products, from mobile phones to cars, are starting to exploit the advantages of continuous collection of product data. For example, connected cars can collect diagnostic data such as fuel efficiency and energy consumption data, whereas telecom equipment can collect performance data such as real-time bandwidth, restarts, outages, upgrade success rate, etc. Therefore, although the first area of post-deployment data collection can be found in online services such as web technologies, the techniques can be applied to any software product that is connected to the Internet for data access and retrieval. This includes software-intensive embedded products intended for a mass market for which evolving needs might be difficult to capture during pre-deployment phases.

12.3 Research Sites and Method

12.3.1 Research Sites

This chapter presents research based on a multiple case study conducted at three software development companies. Today, all the companies are collecting large amounts of data from the products they release to customers.

Company A is a provider of telecommunication systems and equipment, communications networks, and multimedia solutions for mobile and fixed network operators. The company has a number of post-deployment data collection mechanisms in place and is currently collecting data related to system operation and performance. For the purpose of this study, we met with key stakeholders at two company sites in two different countries. The first site is involved in the development and maintenance of nodes within the 3G networks, and at this site, we conducted group interviews with a total of 19 people, including product managers, project managers, support managers, product specialists, integration leaders, developers, and system architects. Also, a workshop was held in which we met with all people from the group interview, as well as with a few additionally invited managers, to discuss and confirm our findings. The second site is involved in the

development, supply, and support of media gateways for mobile networks. At this site, we conducted a group interview with six people involving two department managers, a support manager, a senior specialist, a product manager, and an integration leader.

Company B is a manufacturer and supplier of transport solutions for commercial use. The company has a number of sophisticated data collection mechanisms implemented in their products, and the majority of the data they collect is diagnostic data. For the purpose of this study, we met with two attribute leaders, two developers, and one software expert focusing on software process improvement. In addition, we met with a group of managers and developers focusing on the human machine interface of the vehicles.

Company C is world leading in network video and offers products such as network cameras, video encoders, video management software, and camera applications for professional IP video surveillance. The company has a number of post-deployment data collection mechanisms in place in their products. The data they collect is primarily performance data related to operational use of their products. For the purpose of this study, we conducted five group interviews in which we met with developers, testers, system architects, product owners, project managers, and product specialists. In total, we met with 44 people.

12.3.2 Research Method

Our study reports on a multiple case study [13]. The main data collection method used was semistructured group interviews with open-ended questions [14]. In total, eight group interviews were conducted. All group interviews were conducted in English and lasted for 2 hours. In total, we have 18 hours of recorded interviews and 58 pages of summarizing notes. During analysis, the summary notes were used when coding the data, and as soon as any questions or potential misunderstandings occurred, the recordings were used to replay the discussion and capture all interview details.

In terms of data analysis, a qualitative grounded theory approach was adopted [15]. In this process, open coding principles were used, and clusters and categories emerged as a result of reading the transcribed data to identify similarities in the respondents' experiences. A problem that has been identified in relation to qualitative research is that different individuals may interpret the same data in different ways [16]. This problem was addressed in two ways. First, the coding processes provide an audit trail of the process by which conclusions are reached. Second, we used a "venting" method, i.e., a process whereby interpretations are discussed with professional colleagues [17]. By sharing notes and by discussing the results of each group interview, we could develop an accurate understanding of the different contexts and explore the research questions guiding this study.

12.4 Post-deployment Data Collection and Usage

Our interviews reveal that huge amounts of post-deployment data are collected in all companies. In company A, data is collected in relation to system operation and performance. Information on restarts, system outage, faults, card re-booting, and upgrade success rate is collected and used for assessing system performance and behavior. In addition, dimensioning data such as CPU load, licenses sold, etc., serve as important input for system configuration and capacity, as well as for producing sale statistics and market assessments. As mechanisms for collecting this data, company A reports on a number of support logs and counters, monitoring systems, customer satisfaction indexes, and tools for collecting and storing trouble reports, trouble tickets, and customer requests. While all respondents agree that post-deployment data is important, they experience difficulties when it comes to getting an overview on what is collected and for what purpose. Typically, statistical analysis and trend analysis are done based on the collected data, and there is the opportunity to learn about system performance and future dimensioning needs. However, while performance data, such as upgrade success and downtime reports, is collected, company A reports on difficulties to use the data. As it seems, customer data is used for troubleshooting and for maintaining the current version of the system but very seldom for improving functionality or as a base for developing new functionality. When asking what the key opportunities with increased usage of post-deployment data would be, the interviewees in company A all emphasize the ability to continuously validate what functionality customers value and to improve requirements prioritization.

In company B, post-deployment data is continuously collected in order to assess system behavior of the vehicle. Performance data such as speed, fuel efficiency, energy consumption, acceleration, road conditions, etc., is collected for evaluation purposes. In addition, diagnostic data such as trouble codes, failure reports, etc., is collected by the electronic nodes in the vehicle in order to help troubleshoot a problem whenever the vehicle is handed in for service. Finally, data is collected in order to fulfill legislation purposes since company B is involved in development of products where safety regulations are immense. All respondents agree that the challenge is not to collect data but to make it useful within the organization. Based on the data collected, data mining techniques are used to learn about system performance. However, while this data is useful for the next version of the product family, it is collected with long intervals and is not used for improving the current version of the product. Also, integrating and visualizing the data are found difficult. In company B, all interviewees see the ability to continuously validate what functionality customers value and especially to optimize customer use of the product, as the two main opportunities with increased usage of post-deployment data.

In company C, post-deployment data is collected for assessing system performance. Data on frames per second, stability, and usage hours is important, as well as configuration data on product models and number of sites. The interviewees find

Table 12.1 Summary of post-deployment data collection and usage and the challenges and opportunities the companies experience

Company	Data collection	Data usage	Challenges	Opportunities
A	System operation and performance data Dimensioning data	Statistical and trend analysis Dimensioning needs Trouble shooting System maintenance	Limited overview over collected data No use of data for improvements and new feature development No understanding of feature usage	To continuously validate what functionality customers value To improve requirements prioritization
B	Diagnostics data Performance data	System performance Troubleshooting	Difficulties in integration and visualization of data No use of data for improving the current version of the system No understanding of feature usage	To continuously validate what functionality customers value To optimize customer use of the product
C	System operation and performance data	Customer requests System support	No understanding of feature usage	To increase delivery frequency of functionality To increase the ability to anticipate future customer needs

post-deployment data useful for answering to customer requests and for system support. However, there are no established techniques for post-deployment data usage. While large amounts of data are generated in the systems, this is not used to systematically improve current versions of the products. As a result, interviewees feel that they have limited knowledge on what features of their products that are used, and they feel that whenever post-deployment data is used, a problem has already occurred. In company C, increasing delivery frequency of functionality and increasing the ability to anticipate future customer needs are regarded as the two key opportunities with increased usage of post-deployment data.

While still in the process of establishing techniques for post-deployment data usage, all companies view this activity as critical for continuous validation of their development efforts. In Table 12.1, we summarize our findings and outline the opportunities the companies foresee.

Our study shows that post-deployment data constitutes an enormous asset for companies. However, data usage is limited, and while all companies report on this data as useful for troubleshooting and support, they recognize that mechanisms for continuous improvement of existing product versions, as well as for innovation of new functionality, are not in place.

12.5 Discussion

Everyone involved in development of a software product has ideas on how to make it better. Typically, these ideas are collected and prioritized during the requirements engineering process. Often, the selection and prioritization of ideas are based on previous experiences and opinions and on predictions by product management [4]. However, with a growing number of products being Internet connected, the opportunity to collect post-deployment data has significantly increased [5]. Due to the online nature of most systems, data can be collected as soon as customers use them, and the cost of collecting data from, and about, the customer is low [4, 18, 19]. These benefits apply to software-intensive embedded products such as telecom equipment and vehicles. With the majority of functionality being software and with the opportunity to be connected to the Internet, these products are now increasingly interesting from a data collection perspective. Therefore, although the first area of post-deployment data collection can be found in online services and Web 2.0 systems [5], the techniques can be transferred to any product that is able to collect and provide data about its real-time usage. From a product development perspective, this is interesting as it opens up for continuous improvement of existing products.

12.5.1 The "Post-deployment Data Usage Pyramid"

On the basis of qualitative interviews, we see that usage of this data is limited. In all companies, data related to system operation and performance, and diagnostics, is collected. This data is used for troubleshooting and support activities when the system experiences a problem or an error. According to our interviewees, this reflects a re-active use of the data and difficulties in using the data for more advanced purposes such as for understanding how individual features are used, for improvement of existing features, or for development of new features. In Fig. 12.1, this shortcoming is illustrated in the "post-deployment data usage pyramid."

In our model, we outline the different levels of post-deployment data usage, i.e., different purposes for which data is used. At the first level, operational and performance data represents data that helps companies understand how the system is performing, i.e., data generated during real-time use and that is collected in order for companies to get a system-level understanding. At most companies, operational data is collected without a clear purpose of how to analyze and use it, and therefore, primarily a high-level understanding of the system is obtained. At the second level, diagnostic data represents data that is collected with the specific purpose of supporting troubleshooting activities. Here, data is collected for bug fixing and error correction purposes and for providing input for maintenance. At this level, a more systematic collection of data is required, and companies make use of effective

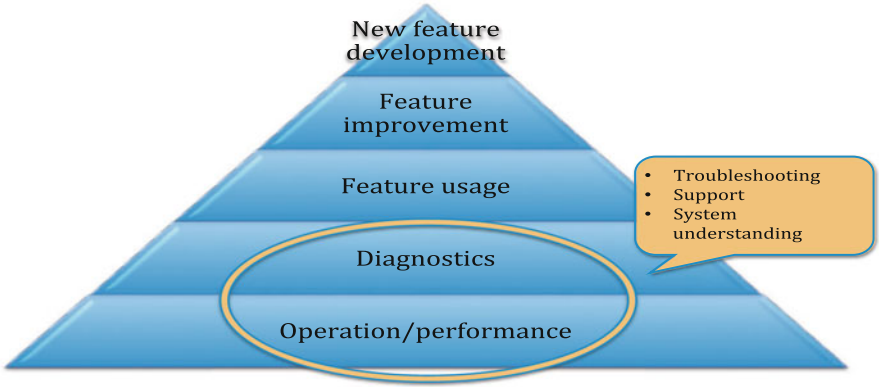


Fig. 12.1 The "post-deployment data usage pyramid"

data storage in order to document and trace troubleshooting and maintenance processes. The third level represents a level at which companies collect data that helps them understand the usage of individual features. In comparison to the high-level system understanding that is provided by collection of operational data, this level requires mechanisms and tools that allow for a more sophisticated data analysis in which usage patterns of specific features can be discerned. At the two most advanced levels, data is collected in order to support continuous improvement of current functionality and as a basis for development of new features. To achieve these levels, advanced development practices that allow for new software functionality to be easily tested and integrated are required.

In showing that usage of post-deployment data is limited, our study confirms previous research [4, 5, 18, 19] in that even though collection of post-deployment data is increasing, there is a range of opportunities still to explore. Our interviews show that the post-deployment data that is collected, i.e., the operation and performance, and diagnostics data as illustrated in Fig. 12.1 support primarily three purposes, i.e., (1) pre-development input, (2) troubleshooting input, and (3) - system-level understanding. In Table 12.2 we summarize the different purposes that post-deployment data supports.

Table 12.2 A summary of the different purposes that post-deployment data supports

Purpose of data collection	Description
Pre-development input	Post-deployment data is used as input to the next pre-development phase, but *not* for improvement of existing system versions
Troubleshooting input	Post-deployment data is used for troubleshooting and support activities, but not for innovation of new functionality
System-level understanding	Post-deployment data provides a system-level understanding of operation and performance, but does *not* provide insight in individual feature usage

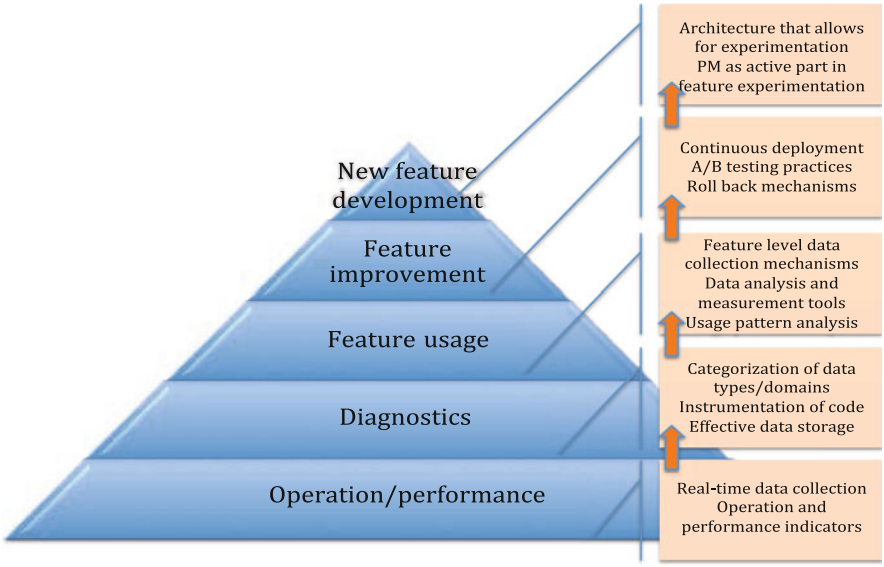


Fig. 12.2 The "post-deployment data usage framework"

12.5.2 The "Post-deployment Data Usage Framework"

As a result of our study, we propose a framework that supports companies interested in advancing their usage of post-deployment data (Fig. 12.2). Based on insights acquired during our study, as well as on our previous work on how to advance beyond agile development practices [12], our framework suggests mechanisms (see the boxes to the right in the figure) that are needed for climbing the different levels and move towards more advanced data usage. These mechanisms are related to organizational processes and development practices and will allow a company to use post-deployment data for more advanced purposes such as an understanding of feature usage, improvement of existing features, and development of new features.

Conclusions
In this paper, we explore collection and usage of post-deployment product data. We highlight the existing limitations in post-deployment data usage and the untapped resource that post-deployment product data remains. Based on a multiple case study at three software development companies, we present the following findings:

(continued)

- **Pre-development input**: Post-deployment data is used as input to the next pre-development phase, but *not* for improvement of existing product versions.
- **Troubleshooting input**: Post-deployment data is used for troubleshooting and support activities, but not for innovation of new features.
- **System-level understanding**: Post-deployment data provides a system-level understanding of operation and performance, but does *not* provide insight in individual feature usage.

Finally, we propose a framework in which we outline what development practices and organizational mechanisms that need to be in place for advancing the usage of post-deployment data and advance the development of software-intensive embedded products.

References

1. Fogelström, N.D., Gorschek, T., Svahnberg, M., Olsson, P.: The impact of agile principles on market-driven software product development. J. Softw. Maint. Evol.: Res. Pract. **22**, 53–80 (2010)
2. Highsmith, J., Cockburn, A.: Agile software development: The business of innovation. Software Management, September, IEEE Computer Society Press Los Alamitos, CA, USA, pp. 120–122 (2001)
3. Kohavi, R., Longbotham, R., Sommerfield, D., Henne, R.M.: Controlled experiments on the web: survey and practice guide. Data Min. Knowl. Disc. **18**(1), 140–181 (2009)
4. Bosch, J.: Building products as innovations experiment systems. In: Proceedings of 3rd International Conference on Software Business, 18–20 June, Cambridge (2012)
5. Bosch, J., Eklund, U.: Eternal embedded software: Towards innovation experiment systems. In: Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change, pp. 19–31. Springer, Berlin (2012)
6. Mishra, D., Mishra, A.: Complex software project development: agile methods adoption. J. Softw. Maint. Evol.: Res. Pract. **23**, 549–564 (2011)
7. Abrahamsson, P., Conboy, K., Wang, X.: 'Lots done, more to do': the current state of agile systems development research. Eur. J. Inf. Syst. **18**(4), 281–284 (2009)
8. Larman, C.: Agile and Iterative Development: A Manager's Guide. Addison-Wesley, Boston, MA (2004)
9. Beck, K.: Embracing change with extreme programming. Computer **32**(10), 70–77 (1999)
10. Bennett, K.H., Rajlish, V.T.: Software maintenance and evolution. In: Proceedings of the 22nd International Conference on Software Engineering (ICSE), Limerick, Ireland, 4–11 June, (2000)
11. Larman, C., Vodde, B.: Scaling Lean & Agile Development: Thinking and Organizational Tools for Large-Scale Scrum. Addison-Wesley, Boston, MA (2008).
12. Olsson, H.H., Alahyari, H., Bosch, J.: Climbing the "Stairway to Heaven": A multiple-case study exploring barriers in the transition from agile development towards continuous deployment of software. In: Proceedings of the 38th Euromicro Conference on Software Engineering and Advanced Applications, 5–7 September, Cesme, Izmir, Turkey (2012)
13. Walsham, G.: Interpretive case studies in IS research: nature and method. Eur. J. Inf. Syst. **4**, 74–81 (1995)

14. Runesson, P., Höst, M.: Guidelines for conducting and reporting case study research in software engineering. Empir. Softw. Eng. **14**, 131–164 (2009)
15. Corbin, J., Strauss, A.: Basics of Qualitative Research: Grounded Theory Procedures and Techniques. Sage, California (1990)
16. Kaplan, B., Duchon, D.: Combining qualitative and quantitative methods in IS research: a case study. MIS Q. **12**(4), 571–587 (1988)
17. Goetz, J., LeCompte, D.: Ethnography and Qualitative Design in Educational Research. Academic, Orlando (1984)
18. Olsson Holmström, H., Bosch, J.: Post-deployment data collection in software-intensive embedded products. In: Proceedings of the 4th International Conference on Software Business, 11–14 June 2013, Potsdam, Germany (2013)
19. Olsson Holmström H., Bosch J.: Towards data-driven product development: a multiple case study on post-deployment data usage in software-intensive embedded systems. In: Proceedings of the Lean Enterprise Software and Systems Conference (LESS), 1–4 December, 2013, Galway, Ireland (2013)

Chapter 13
The HYPEX Model: From Opinions to Data-Driven Software Development

Helena Holmström Olsson and Jan Bosch

Abstract While innovation, such as development of new features, is critical for any organization, it is hard to get right. In both our case companies, the selection of ideas is usually driven by previous experiences, and very often the process becomes politicized and based on peoples' opinions. To address this, we present the Hypothesis Experiment Data-Driven Development (HYPEX) model. Our model is an alternative development process that helps companies shorten the feedback loop to customers. The model supports companies in running feature experiments and advocates development of small parts of features that are continuously evaluated with customers. In our study we validate the model in two software development companies. Although the companies involved in the study have not yet completed a full experiment cycle, we see that feature experiments are beneficial for improving at least four activities within the companies: (1) data-driven development (the ease of collecting customer feedback allows for a real-time connection between the quantified business goals of the organization and the operational metrics collected from the installed customer base), (2) customer responsiveness (the ease of collecting customer feedback allows product management to respond rapidly and dynamically to any changes to the use of the products, as well as to emerging customer requests), (3) R&D efficiency (the ease of collecting customer feedback gives the development teams a real-time goal and metrics to strive for and provides focus for their work), and (4) R&D accuracy (the ease of collecting customer feedback enables the development teams to align their efforts with what the customers appreciate the most). The HYPEX model is a development process that helps software development companies move away from building large chunks of functionality with little feedback from customers and instead continuously validate with customers that the functionality under development is of value to customers.

H.H. Olsson (✉)
Department of Computer Science, Malmö University, Malmö, Sweden
e-mail: helena.holmstrom.olsson@mah.se

J. Bosch
Department of Computer Science and Engineering, Chalmers University of Technology, Gothenburg, Sweden
e-mail: Jan@JanBosch.com

J. Bosch (ed.), *Continuous Software Engineering*,
DOI 10.1007/978-3-319-11283-1_13

13.1 Introduction

In most software development companies, the road mapping and requirements prioritization process are a complex endeavor in which product management experiences difficulties in getting timely and accurate information [1, 2]. The feedback loop from customers is slow, and often there is a lack of mechanisms that allow for efficient customer data collection and analysis. As a result, most companies build large chunks of functionality without getting feedback from customers, and the value of the features they prioritize is not efficiently validated with customers. What often happens is that requirements prioritization becomes an opinion-based and politicized process rather than a data-driven process in which customer data guide future R&D investments [3]. Also, without the opportunity to continuously confirm customer value, there is the risk of lack of alignment of product and customer needs during the road mapping of new software features [2].

To address this problem, we develop the "Hypothesis Experiment Data-Driven Development" (HYPEX) model. Our model is a fundamentally new development process that supports companies in running feature experiments to shorten feedback loops and to continuously validate functionality with customers. While experimentation with customers is a well-established practice in the Web 2.0 and SaaS domain [4], it has not been applied extensively in relation to large-scale development of embedded systems. We validate the model in two software development companies, and we provide evidence on how feature experiments increase the opportunity for data-driven software development.

13.2 Background

13.2.1 From Traditional to Data-Driven Development

In our previous research [5], we outline the typical evolution path that software development companies follow when moving from traditional development to customer data-driven development. We named the model the "Stairway to Heaven," and it has proven useful for identifying what actions companies need to take to advance its practices [5, 6]. In our model, the final stage is where the company realizes that frequent deployment of software to customers can be used for continuous testing of new features, as well as for optimization of existing features. This is where large-scale embedded systems companies start realizing that many of the benefits that have so far been exclusive to Web 2.0 and SaaS companies have become available also to them [3, 5, 7]. At this step, the entire R&D organization responds and acts based on instant customer feedback, and deployment of software is seen as a way of validating what the customer needs. Recently, the concept of R&D as experiment systems has been defined as an experiment-centric approach to product development with the purpose of

accelerating innovation through systematic and continuous collection of customer feedback [3, 7]. Common for experiment systems is that requirements evolve in real-time based on customer usage data, instead of being frozen early based on the opinions of product management [3].

13.2.2 Data-Driven Software Engineering Practices

Data-driven practices are not new to the software engineering field, and there are a number of practices that reflect the sincere interest to find metrics that continuously validate success in terms of customer value. One example is data-driven software engineering [8]. Here, continuous collection of data is used to understand the successful development of software systems [8]. During the development cycle, different metrics related to product quality are continuously collected. The goal is to use such metrics to make estimates of post-release failures early in the software development cycle, as well as during the implementation and testing phases. Such estimates can help focus testing, code, and design reviews and affordably guide corrective actions and decision-making activities.

Another example is value-based software engineering, which is a practice focused on increasing a company's business value by improving the economic efficiency of the software they develop [9]. This is achieved by continuous gathering of feedback on the current expenses and the expected profits of a project and then making adjustments and corrective actions based upon these values. Value-based software engineering models are meant to show how each project will impact the value of the business by continuous collection and analysis of relevant data points [10]. Only components and software projects that are deemed economically viable will be developed [9].

Finally, and of particular interest for our research, A/B testing is possibly the most fundamental of mechanisms to capture product use and customer behavior [3]. A/B testing is concerned with presenting one version of the software to some customers and another version to other customers. The behavior of both groups is compared in order to determine which version (A or B) that leads to more desirable outcomes for the company providing the product. In online offerings, the notion of A/B testing is used extensively as a mechanism to measure customer behavior and determine what version customers appreciate the most [4]. As a result, these systems evolve continuously, and whereas earlier this was achieved by yearly releases of new software, we see a trend where continuous testing of new, innovative functionality in deployed systems is increasingly applied in online systems [3]. In this way, requirements evolve in real time based on data collected from customers, instead of being frozen early based on the opinions of product management.

13.3 Research Method

13.3.1 Case Study Research

This study builds on a multi-case study in which the authors of the paper interacted with two software development companies. Case study research focuses on providing deeper understanding of a particular phenomenon and is typically used to explore contemporary phenomena in its natural context [11, 12]. Since research in software engineering is to a large extent a multidisciplinary area aimed at investigating how development, implementation, use, and maintenance of software are conducted, we found the case study approach appropriate for our study [11]. Our study involves two software development companies, referred to as company A and B.

Company A is a software company specializing in navigational information, operations management and optimization solutions, crew and fleet management solutions, and flight training products and services. In company B, the feature experiment is part of a strategic move from manual planning to optimization with formalized business metrics. As part of the experiment, a new feature is developed that allows the user to tune input parameters in correspondence to the business metrics and to send new optimization runs based on these. As part of the experiment, the company collects metrics on how their customers interact with the new feature. For the company, the desired outcome is a situation in which the users launch more optimization jobs based on high-level, and agreed-upon, business metrics.

Company B is world leading in network video and offers products such as network cameras, video encoders, video management software, and camera applications for professional IP video surveillance. In company B, the feature experiment concerns remote access of cameras. While remote access has become one of the most critical features in surveillance, it is also one of the most difficult to implement. For the company, the desired outcome is to provide users with a solution that allows for easier access to their cameras regardless of their network environment. During the experiment, data is collected continuously by pattern matching in log files. As a start, the company has chosen a limited set of target users with whom the experiment will be initially run.

13.3.2 Data Collection and Analysis

Our research is based on a mix of group interviews and workshops at the two case companies. The group interviews worked as input to our understanding of each company and the problems they experience. Following these interviews, workshops sessions were held to initiate feature experiments with the purpose to help the

companies shorten feedback loops to customers. In total, five group interviews and three workshops were held with key stakeholders from the companies.

In company A, we have so far conducted one group interview and one workshop including key stakeholders such as chief architects, project managers, and software developers. Additional activities are planned, and the evaluation of the experiment will involve a number of interviews. For the purpose of this study, company A is an example of a software company engaging in its first feature experiment with customers.

In company B, we conducted five group interviews including key stakeholders in the organization. In addition to these interviews, two workshop sessions were held. In these workshops, project managers as well as developers and architects were present in order to cover all aspects of the intended feature experiment. For the purpose of this study, company B is an interesting example of a company involved in large-scale development of embedded systems and starting its first feature experiment.

All group interviews and workshops were in English and lasted for 2–3 h. In addition to the interview notes, all interviews were recorded in order for the researchers to have a full description of what was said [13]. Each interview was transcribed, and the transcriptions were shared between the researchers to allow for further elaboration on the empirical material. During analysis, all transcribed interviews were carefully read with the intention to identify recurring elements and concepts. During the workshops, notes were taken to capture the discussions, and white board illustrations were documented using a camera. Also, presentations held by company representatives were shared with the researchers to help create a common understanding for the feature they selected for their experiments, as well as for the organizational units the experiment would involve.

13.4 Findings

The focus of this paper is to find mechanisms that help companies shorten feedback loops to customers and allow companies to continuously validate software functionality with customers. In the case study companies, we identified a number of problems that emerge as a result of slow feedback loops. Below we discuss these problems in more detail before proposing a model that helps companies address these problems.

Feature Development Without a Confirmed Value for Customers: While there is significant investment and effort put into new feature development, there are no established practices for validating whether new features correspond to customer needs. For product management, the lack of confirmation from customers leads to a situation in which decision making and prioritization are made based on opinions rather than data, and there is the risk that the prioritizations are not aligned with customer needs.

Politicized Prioritization Process: Our case companies experience the feature prioritization process as highly politicized. Due to lack of customer data, there is no efficient way to determine whether a feature will generate customer value. Typically, the selection process of what features to develop is driven by previous experiences and beliefs. Also, it is often the opinions of the more senior people in the organization, rather than data, that have the greatest impact on the selection and prioritization process.

Lack of Clarity of Feature Content: The lack of customer feedback results in development teams guessing what content that should go into a feature. This results in a situation where development teams experience frustration when not knowing what generates value for customers. From an organizational perspective, the situation causes inefficiency as well as an expensive development organization.

Lack of Alignment of Product with Customer Needs: Our case companies recognize the risk of having a product that deviates from what their customers need. While an individual feature can always be improved, the cost of having a product that doesn't align with customer needs is a major risk.

13.5 The HYPEX Model

In response to the problems mentioned above, we developed the Hypothesis Experiment Data-Driven Development (HYPEX) model. The model is an alternative development process model that helps companies shorten feedback loops to customers. The model is presented in Fig. 13.1.

Feature Backlog Generation: The first practice in the HYPEX model is the generation of features that may potentially bring value to customers. Product

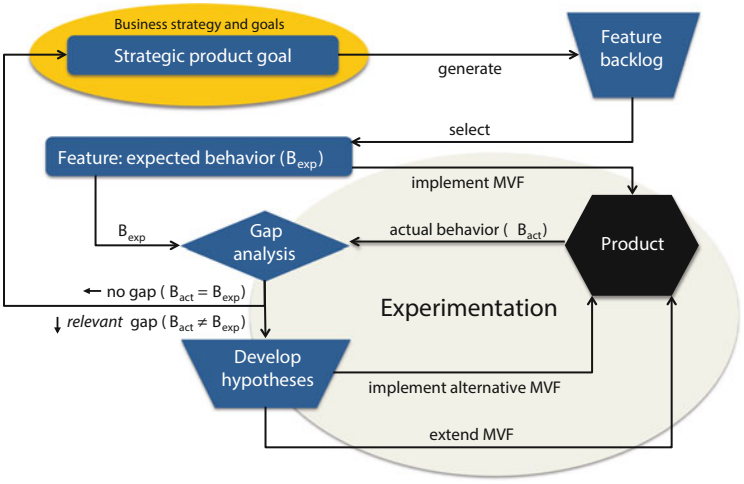


Fig. 13.1 The Hypothesis Experiment Data-Driven Development (HYPEX) model

management and development teams base the generation of features on strategic business goals and their understanding of customer needs. The features are entered into a feature backlog. At this point, the feature backlog is a list of potential features that may or may not be selected for implementation.

Feature Selection and Specification: The selection of what feature to experiment with can be done based on a number of reasons including (1) it addresses an area of functionality with a big gap between expected and actual behavior, (2) it concerns development of new functionality in an area where there is little previous experience, or (3) it concerns development of new functionality in an area where there is multiple alternatives of implementing the feature. Once the feature is selected, the expected behavior of the feature is defined, i.e., how the feature adds value to a customer and how it supports strategic business goals. The definition of the expected behavior allows for quantitative analysis and for an organizational dialogue about feature behavior.

Implementation and Instrumentation: The third practice is concerned with implementing and instrumenting the first "slice" of the feature. In the model we refer to this as minimal viable feature (MVF) with the intention to identify the smallest possible part of a feature that adds value to a customer. A feature is implemented in multiple iterations starting from the most important functionality. Instrumentation will allow the organization to measure the actual behavior of the feature when in the hands of a customer. Once the functionality is deployed, data collection starts allowing the team to collect statistically relevant data about the actual behavior.

Gap Analysis: During gap analysis, the expected behavior is compared with the actual behavior to determine whether the current implementation of the feature is sufficient to achieve the expected behavior. In case the gap is sufficiently small, the development team finalizes the feature. In case there is a significant gap, the team starts developing hypotheses to explain the gap. The third outcome is that the team decides to abandon the feature altogether as the expected benefit from the feature is not achieved and the current implementation may show no or even a negative effect on the actual behavior. The gap analysis is central for shortening the feedback loop. Rather than guessing the benefits of a feature, the organization gets data on its behavior. As a result, informed decisions can be taken instead of opinion-based and politicized decisions.

Hypothesis Generation and Selection: If the team identifies the gap in expected and actual behavior and decides to continue development, the next step is to generate hypotheses that explain the gap. There are two main categories of hypotheses. The first is that the slice of the feature implemented is not sufficient for the customer to experience the benefits. In this case, the MVF is extended so that more accurate metrics can be collected. The second category is concerned with the belief that an alternative implementation of the MVF will yield a different outcome.

Alternative Implementation: In case the outcome of the previous step is that the implementation of the MVF does not meet the needs of customers, the team decides to build an alternative implementation. This is often referred to as A/B testing [2]. In deployed embedded systems, a first version (A) is deployed and data

is collected. Subsequently, version (B) is deployed and again data is collected. Based on the difference in actual behavior between the two versions, the hypothesis can be validated. Once sufficient data has been collected, the team returns to the gap analysis.

13.6 Industrial Experiences

In our study, each company engaged in feature experiments with the intention to shorten feedback loops and increase their understanding of feature usage. So far, both case companies have adopted the first three practices, i.e., feature backlog generation, feature selection and specification, and implementation and instrumentation. They are both in the middle of running their first experiment, and we have not yet been able to validate the full experiment cycle with them. However, although the companies haven't benefitted from a full experiment cycle yet, there are already a number of interesting lessons learned in relation to the problems identified earlier in this chapter.

Both companies share with us that only by initiating the discussion about feature experiments, and by starting the process of feature selection and specification, they benefit from an improved understanding of why certain organizational assumptions exist. To select a feature, and to specify the expected behavior when in the hands of a customer, adds to the organizational awareness of why certain problems exist. As a result, both companies now have a clear understanding for the desired outcome of their experiments, as well as ideas on how to conduct the gap analysis and hypothesis generation. Also, the discussion about what potential features to work with, and the definition about the expected behavior of these, reflects the company culture as well as the current, but insufficient, understanding of their customers. In helping different stakeholders to pinpoint existing problems or uncertainties related to feature usage, the HYPEX model has already improved the situation in both companies, and they are both eager to proceed with the full experiment cycle. The benefits and challenges that the companies have experienced so far are summarized in Table 13.1.

Table 13.1 Lessons learned in company A and B

	Benefits	Challenges
Company A and B	Improved communication between organizational units Rewarding definition of metrics Improved understanding of how to collect data Improved understanding of what data to collect Improved understanding of quality issues Better understanding of the benefits with early user involvement	Establish an "experiment mindset" among employees Identify a feature to experiment with Develop smaller parts of a feature Identify customers to work with Define metrics and what data to collect Customer support and training

Conclusions
While innovation, such as development of new features, is critical for any organization, it is hard to get right. In both our case companies, the selection of ideas is usually driven by previous experiences, and very often the process becomes politicized and based on peoples' opinions. To help solve this situation, we present the Hypothesis Experiment Data-Driven Development (HYPEX) model. Our model is an alternative development process that helps companies shorten the feedback loop to customers. The model supports companies in running feature experiments and advocates development of small parts of features that are continuously evaluated with customers.

In our study, we validate the model in two software development companies. Although the companies involved in the study have not yet completed a full experiment cycle, we see that feature experiments are beneficial for improving the following activities within the companies:

- **Data-driven development**: The ease of collecting customer feedback allows for a real-time connection between the quantified business goals of the organization and the operational metrics collected from the installed customer base.
- **Customer responsiveness**: The ease of collecting customer feedback allows product management to respond rapidly and dynamically to any changes to the use of the products, as well as to emerging customer requests.
- **R&D efficiency**: The ease of collecting customer feedback gives the development teams a real-time goal and metrics to strive for and provides focus for their work.
- **R&D accuracy**: The ease of collecting customer feedback enables the development teams to align their efforts with what the customers appreciate the most.

The HYPEX model is a development process that helps software development companies move away from building large chunks of functionality with little feedback from customers. When adopting the HYPEX practices, companies allow for continuous validation with customers resulting in data-driven software development practices.

References

1. Olsson, H.H., Bosch, J.: Post-deployment data collection in software-intensive embedded products. In: Proceedings of the 4th International Conference on Software Business, Potsdam, 11–14 June 2013
2. Olsson, H.H., Bosch, J.: Towards data-driven product development: A multiple case study on post-deployment data usage in software-intensive embedded systems. In: Proceedings of the Lean Enterprise Software and Systems Conference (LESS), Galway, 1–4 December 2013

3. Bosch, J.: Building products as innovations experiment systems. In: Proceedings of 3rd International Conference on Software Business, Cambridge, 18–20 June 2012
4. Kohavi, R., Crook, T., Longbotham, R.: Online experimentation at Microsoft. In: van der Putten, P., Melli, G., Kitts, B. (eds.) Proceedings of the Third International Workshop on Data Mining Case Studies, held at the Fifteenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining in Paris, France, pp. 11–23 (2009)
5. Olsson, H.H., Alahyari, H., Bosch, J.: Climbing the "Stairway to Heaven": A multiple-case study exploring barriers in the transition from agile development towards continuous deployment of software. In: Proceedings of the 38th Euromicro Conference on Software Engineering and Advanced Applications, Cesme, 5–7 September 2012
6. Olsson, H.H., Bosch, J.: Towards R&D as innovation experiment systems: A framework for moving beyond Agile software development. In: Proceedings of the IASTED, pp. 798–805 (2013)
7. Bosch, J., Eklund, U.: Eternal embedded software: Towards innovation experiment systems. In: Proceedings of International Symposium on Leveraging Applications, Crete, 15–18 October 2012
8. Bird, C., Murphy, B., Nagappan, N., Zimmermann, T.: Empirical software engineering at Microsoft research. In: Proceedings of CSCW, Hangzhou, 19–23 March 2011
9. Biffl, S., Aurum, A., Boehm, B., Erdogmus, H., Grunbacher, P. (eds.): Value-Based Software Engineering. Springer, Berlin Heidelberg (2006)
10. Madachy, R.: Integrated modeling of business value and software processes. In: Proceedings of the International Software Process Workshop, SPW 2005, Beijing, 25–27 May 2005. Revised Selected Papers, pp. 389–402. Springer, Berlin (2006)
11. Runesson, P., Höst, M.: Guidelines for conducting and reporting case study research in software engineering. Empir. Softw. Eng. **14**, 131–164 (2009)
12. Yin, R.K.: Case Study Research. Design and Methods, 3rd edn. Sage, London (2003)
13. Walsham, G.: Interpretive case studies in IS research: nature and method. Eur. J. Inf. Syst. **4**, 74–81 (1995)

Part V
Organizational Performance Metrics

Whereas the earlier parts discussed specific steps on the Stairway to Heaven, this part is concerned with developing effective quantitative techniques to assess the performance of a development organization. There are two chapters in this part. The first chapter presents a method for developing product and organizational performance profiles. These profiles can be used to quantify properties of products before release as well as to quantify performance of software development processes. The approach is illustrated with case studies from several Software Center companies. The second chapter focuses on the challenge of keeping an automated measurement system operational in a constantly changing environment and introduces the concept of self-healing. The approach is illustrated through the infrastructure developed at Ericsson.

Chapter 14
Profiling Prerelease Software Product and Organizational Performance

Vard Antinyan, Miroslaw Staron, and Wilhelm Meding

Abstract *Background*: Large software development organizations require effective means of quantifying excellence of products and improvement areas. A good quantification of excellence supports organizations in retaining market leadership. In addition, a good quantification of improvement areas is needed to continuously increase performance of products and processes.

Objective: In this chapter we present a method for developing product and organizational performance profiles. The profiles are a means of quantifying prerelease properties of products and quantifying performance of software development processes.

Method: We conducted two case studies at three companies—Ericsson, Volvo Group Truck Technology, and Volvo Car Corporation. The goal of first case study is to identify risky areas of source code. We used a focus group to elicit and evaluate measures and indicators at Ericsson. Volvo Group Truck Technology was used to validate our profiling method.

Results: The results of the first case study showed that profiling of product performance can be done by identifying risky areas of source code using combination of two measures—McCabe complexity and number of revisions of files. The results of second case study show that profiling change frequencies of models can help developers identify implicit architectural dependencies.

Conclusions: We conclude that profiling is an effective tool for supporting improvements of product and organizational performance. The key for creating useful profiles is the close collaboration between research and development organizations.

V. Antinyan (✉) • M. Staron
University of Gothenburg, Gothenburg, Sweden
e-mail: vard.antinyan@gu.se; miroslaw.staron@gu.se

W. Meding
Ericsson Software Research, Ericsson AB, Sweden
e-mail: wilhelm.meding@ericsson.com

J. Bosch (ed.), *Continuous Software Engineering*,
DOI 10.1007/978-3-319-11283-1_14

14.1 Introduction

Continuous assessment of product and development performance is a means to support developers in visualization of product status and proactive decision making. In modern software development organizations, this assessment process is usually a complex activity. Glass [1] observed that for every 25% increase in problem complexity, there is 100% complexity increase in software solution. The assessment of contemporary software products is difficult because there are no explicit properties of software which can be directly used to quantify excellence, as unlike other products software products are intangible and require visualization.

The focus of our research project in Software Center is to identify and develop methods and tools for profiling excellence of software products together with the collaborating companies. The goal of this chapter is to present a process for profiling prerelease software product performance. This goal has been accomplished by conducting action research at Ericsson, Volvo Cars Corporation, Volvo Group Truck Technology (GTT), and Saab Electronic Defense Systems. By collaborating closely with industrial partners, we developed different profiling tools and evaluated them for industrial use. The results show that close collaboration with developers facilitates creating useful profiling tools. Two profiling tools that are developed in companies are presented in this chapter.

The rest of the chapter is organized as follows: firstly, we introduce the concept of profiling in the context of large software development organizations; next we present two case studies about prerelease product and process profiling; and finally, we describe the experiences and learning of companies and draw conclusions from this chapter.

14.2 Profiling

To make quick and optimal decisions, managers, architects, designers, and developers need to have good insights on how the developed product or development processes are implemented and how they evolve over time. Mere subjective estimates and opinions might be good enough for small application's development, but for large complex products, quantitative measures are required. Example of measures for profiling could be the ratio of executed tasks and planned tasks, system testing queue over time, throughput trend, architectural dependencies between components, complexity of code, etc.

Generally "assessment" and "evaluation" are widely used by researchers when it comes to discussing product excellence—for example, when conducting benchmarking [2, 3]. However, these concepts are focused on the evaluation of software products. The terms "assessment" or "evaluation" are intended to determine the degree of excellence of a particular entity; *examples of entity can be a single source code function, a model, architecture of product*, etc.

Fig. 14.1 Difference of assessment and profiling of source code

The term *profiling* opens a different perspective when considering about product excellence; it is a neutral concept and regards an entity as a composition of elements. Such examples can be software code composed by source code functions, software architecture composed by architectural components, etc. When profiling the excellence of code instead of merely assessing how good the code is, the functions are assessed and presented in one picture. In Fig. 14.1 we can see that the source code can be assessed to be good (left-hand picture) or profiled in terms of a defined criterion for the excellence of functions (right-hand picture). For example, if the criterion is the maintainability of functions, then three levels of maintainability may be defined and represented by colors; in Fig. 14.1 red represents functions that are hard to maintain, orange color represents functions with moderate maintainability, and green represents functions that are easy to maintain (right-hand picture).

Generalizing we can state that profiling the excellence of an entity is a representation of elements comprising that entity, where the assessments of all elements are available. Although each function can be assessed separately, the essence of profiling assumes that the whole source code, its functions, and their assessments are represented just in one picture. This property is opening a new dimension for evaluation, which is the comparison of functions in one picture or comparison of two products' source code with equivalent representation.

In this paper we define profiling as:

Definition 1 Profiling prerelease software product performance is the measurement and representation of such properties of product, which enables evaluation of that product's excellence before delivery.

This definition emphasizes profiling related to prerelease product properties (e.g., internal quality attributes such as complexity) and post-release success of the product (excellence).

We define profiling organizational performance in similar terms:

Definition 2 Profiling software organizational performance is the measurement and representation of such properties of organizational processes that enables evaluation of organizational performance in product development.

According to definitions, profiling must enable evaluation of excellence. An example of profiling is measuring size, complexity, trend of reported defects, and visualizing them for designers so that problem areas are easily visible.

14.3 Profiling Prerelease Product and Process Performance in Large Software Development Organizations

The software development companies, which we collaborate with, use Lean/Agile principles to assure quick response on customers' requirements. This kind of development usually comprises diverse activities with complex tasks as the product itself is big and complex.

Developing these products is characterized by such challenges as mixed long-term planning for main release, short time planning for service releases, distributed decision making by software development teams, communication between teams, or multisite development. Figure 14.2 presents an overview on how the functional requirements (FR) and nonfunctional requirements (NFR) are prioritized and packaged into work packages by product management (PM), then systemized by system management (SM), and at last implemented and tested by design management (DM) and test teams. Each team delivers their code into the main branch.

Before the release, the development teams are concerned with how good the developed artifacts are and how well the development processes are carried out. Prerelease product profiling is concerned with representing the excellence of developed artifacts and development processes. Improvements of artifacts and processes have a twofold effect: decreasing internal development costs and efforts and implicitly ensuring better quality of released software. The ultimate goal of development teams is to release a product that is complete by functionality and fulfills all the requirements of quality (reliability, usability, efficiency, etc.). Therefore, prerelease improvements of software artifacts and well-designed processes create a high likelihood for a high-quality product.

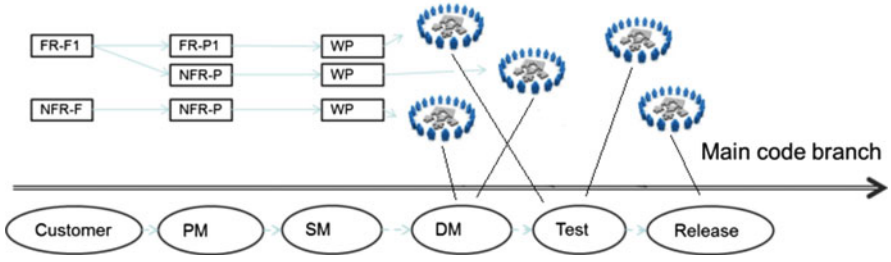


Fig. 14.2 Feature development in Lean/Agile methods

14.4 Establishing the Process of Profiling

In this section we define the process of profiling and provide examples. Before starting the profiling process, it is important to consider what exactly should be profiled and what should be achieved by that—i.e., elicit the information need for the profile [4]. In order to develop a profile of the excellence of prerelease software, the profiling process can be designed as follows:

1. *Identify measurable properties of the product artifacts that allow the assessment of the product's excellence*: Well-known artifacts are software requirements specification, architecture, components, source code files, functions, etc. Well-known properties are ambiguity of requirements, architectural dependencies, code complexity, etc.
2. *Identify measurement tool and measure the specified properties of the artifact*: Example measures for properties are lines of code (LOC) as size measure, McCabe complexity number as complexity measure, structural fan-out as dependency measure, ambiguity ratings in a scale of one to five for requirements, etc.
3. *Identify the influence of measure on artifact's excellence*, *analytically or empirically established thresholds for measures by which the developers can assess the excellence of the artifact*. For example, a threshold for fan-out as a dependency measure could be seven. If a component has more than seven fan-outs, then it is considered too vulnerable to external changes.
4. *Reduce/optimize the number of measured properties by using statistical methods*: Several properties can be related to each other, or metrics might show weak influence on product excellence. For example, number of statements and number of LOC are showing the same property of code and size; thus, one measure should be used. Another example is that the number of functions in a file has weak influence on maintainability and fault proneness of code so it cannot be used to assess maintainability of code.
5. *If possible*, *combine the remaining independent measures using statistical or mathematical methods in a way that they jointly characterize the excellence of the artifact*: For example, requirements ambiguity and complexity ratings can be mixed to jointly represent the difficulty that developers have in understanding the requirements.
6. *Define thresholds for the joint measure to separate a number of artifacts by their excellence*. For example, if a function has greater than 20 cyclomatic number and is called by other functions more than 50 times, then it is considered to be a badly designed function.
7. *Represent the artifact as it is composed by elements*, *where elements are assessed by combining the measure and thresholds*: For example, the source code (artifact) of product with its files (elements) can be represented by measured sizes (characteristic) of files. Files having more than 1,000 (threshold) LOC (measure) are considered big (assessment); the rest are considered normal (assessment).

Before implementing these steps, there are two essential considerations about the quality of measures:

- *Accuracy*: How accurately the measure counts the property of an artifact?
- *Appropriateness*: Does the measure support in decision making?

In our previous work [5], we showed that effective use of measures, as support for decision processes, requires about 20 measures at the top management level.

The risk of developing wrong profile for an artifact is high as the developers not always know what selected measures actually show and how profound basis the established thresholds have. That is why the researchers in the field of software engineering should work beside practitioners in order to guarantee that measurement methods and tools have scientific basis.

In the next two sections, we are giving an overview of how two example measures are developed and selected at large software development companies. One measure is designed for profiling prerelease product performance; the second one is designed to profile the development process.

14.5 Case Study: Developing Risk Profile of Software Code

An action research project was conducted at two companies, Ericsson AB and Volvo GTT, with the aim to create a profile that visualizes the risk of source files. In this context we define the code to be risky if it is fault prone, potentially difficult to maintain, or difficult to manage. As result of conducted project, we developed a method and tool that locate risky source code files. An overview on how the method was created and its usage is presented beneath.

The profiling process starts with identification of code properties that should be measured. According to *Definition 1* these properties must enable assessment and representation of product excellence. In our case, a specific measure of excellence is the riskiness of the code. The more risk-free, the better excellence the code has. Identifications of these properties should have either scientific or intuitive basis. Thus, we measured properties of code that:

- Are manifested in literature to have influence on fault proneness or maintainability of code.
- Are confirmed by many experienced developers to have correlation with their difficulty of managing code.

The measured properties are divided into categories and presented in Fig. 14.3. The Δ letter in the figure indicates the change of the metric over a specified time interval.

We followed action research principles for measurement and selection of metrics. Close collaboration with a reference group of designers at Ericsson, which was administered by the line manager of developed product, allowed us to discuss intermediate results. During the period of 8 months, we conducted analyses and

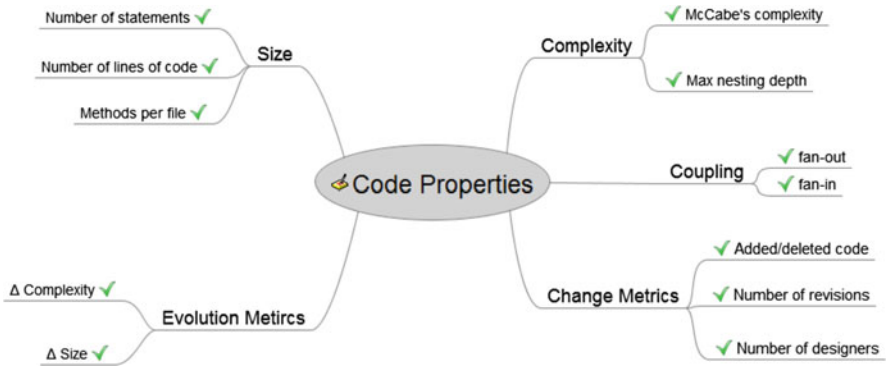


Fig. 14.3 Measured properties

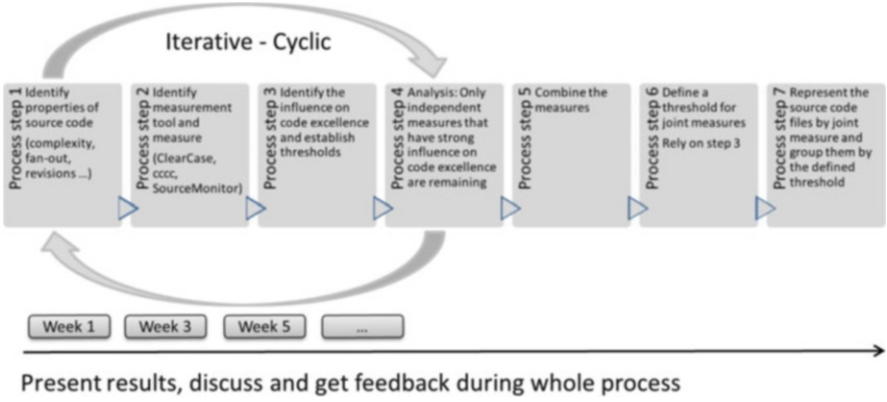


Fig. 14.4 Overview of profiling process carried out at Ericsson

met biweekly with reference group to get feedback on presented results. Figure 14.4 illustrates the applied research method that is compliant to the process described in Sect. 14.4.

The more detailed information of what research activities have been carried out, what results were obtained and presented in each step, and the feedback of designers are presented in Table 14.1.

The table illustrates the short feedback loops and changes in the focus of the project/profile over time—aligned with the model presented by Sandberg et al. [6]. While the discussions with designers allowed us to select metrics and understand reasons why certain files can be very complex or much changed, the statistical techniques were used to limit the number of measures. It is common that when having many different measures, some of them might be strongly correlated and, in fact, show the same property of code. For example, if we measure the number of statements and the number of lines of source code files, we can find that they are strongly correlated. The reason is they both measure the same property of

Table 14.1 Biweekly analyses of results and reference group's feedback at Ericsson

Week	Presented	Feedback
1	Size, complexity, and their evolution between two main releases of the product is observed and presented Strong correlation of size and complexity is found	Skip measuring size Measure the evolution of complexity between four main releases to find out if the complexity increases constantly Observe complexity difference between C and C++ code
3	The complexity of the product increases constantly through four releases C code generally contains more complex functions then C++ code	The overall complexity of the product should increase because it is inherent to increasing functionality. Instead it is important to measure if the number of complex functions is increasing
5	The number of complex functions is increasing over development time	Investigate top functions with highest complexity increase. Find patterns associated with complexity increase and trigger action if necessary
7	The causes of top 30 functions with complexity increase are investigated	The causes are discussed. The reference group decides upon how change report policy should be
9	Similar measurements are carried out on code *generated* from models	Develop an initial measurement system for tracking complexity increases. Measure other proposed properties that might have influence on fault proneness and maintainability
11	Number of revisions of source files is measured Clustering technique is introduced	Thus measure the number of designers as they might have stronger influence
13	Number of designers is measured Strong correlation is found with number of revisions	Skip measuring number of designers
15	Correlation of revisions and error reports is moderate positive: Not all files with high revisions are problem	Combine available independent metrics to obtain stronger indicator of problematic files
17	Fan-out, block depth are measured Strong correlation between complexity and fan-out is presented	No particular feedback Fan-out and block depth are skipped
19	Different clustering techniques are applied to combine complexity and revisions. None of them is suitable	No particular feedback Waiting for successful combination of metrics
21	The product of complexity and revisions is proposed as combined metric Two thresholds are defined for distinguishing low-, moderate-, and high-risk-containing files	The method is intuitive and should be evaluated. The reference group decides to conduct 6 weeks evaluation period
23–27	Evaluation and replication of analysis at Volvo Trucks	Evaluation shows that the proposed method is a good metric for giving risk profile of source code files

code and size. With pairwise correlation analyses and discussing the relations of measures with designers, we could reduce the number of measures to two independent ones, thus avoiding redundant measures.

One of the examples of correlation is illustrated in Fig. 14.5. In the figure every dot represents a c function of telecom software.

The scatterplot of functions shows that not all big functions are complex; there are many functions along with "LOC" axis showing zero complexity, and many functions are in the left uppermost corner of graph showing big size and small complexity. Conversely, there is no function in down most side showing that there is no complex function with small size. This means that if we chose size as a measure for risk identification, we might omit the complexity aspect of code, as not all big functions are complex, whereas choosing complexity as a measure, we know that size is also involved in this measure, as complex functions are also big.

Doing similar analysis with all metrics and discussing results with designers, we could reduce the number of measures to two independent ones—McCabes cyclomatic complexity and number of revisions of files. We combined these two metrics to obtain their joint magnitude as an indication of risk. Denoting the risk as the product of complexity and number of revisions of files, we get:

$$\text{Risk} = \text{Complexity} * \text{Revisions}$$

This formula permits to assess the riskiness of files. Then two thresholds were established for risk level. For example, the files that have risk > 400 score are considered very risky, $200 < \text{risk} < 400$ are moderately risky, etc. One simple example of grouping files according to their risk level is shown in Fig. 14.6. The figure presents the risk profile of source code as a composite of files. All the files of product are divided into three groups:

- The round dots on the scatterplot are files with high risk.
- The squares are files with moderate risk.
- The triangles are files with no or little risk.

The established threshold can vary from company to company, depending on how much complexity developers can tolerate, what the number of revisions show, and how many risky files the developers can manage to refactor or test. The thresholds vary, but, as evaluation showed, the fact that files having many revisions and high complexity are risky is not likely to change.

After assessment, we evaluated the method for 6-week period with designers. The method was confirmed to be accurate and relevant for risk assessment. Both designers' feedback and correlation between risk and error reports show that the method is viable in systematic industrial use for risk assessment. After evaluation, the measurement system was developed to give continuous feedback to designers and management on risky files (Fig. 14.7).

During the research, we measured 12 properties of code (Fig. 14.3), identified relevant properties to select, discussed the intermediate results with experts to assure that the research is going to right direction, created the method, and evaluated it.

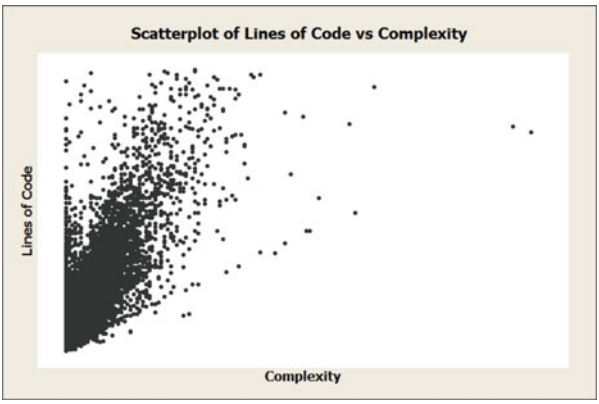


Fig. 14.5 Correlation of complexity and size

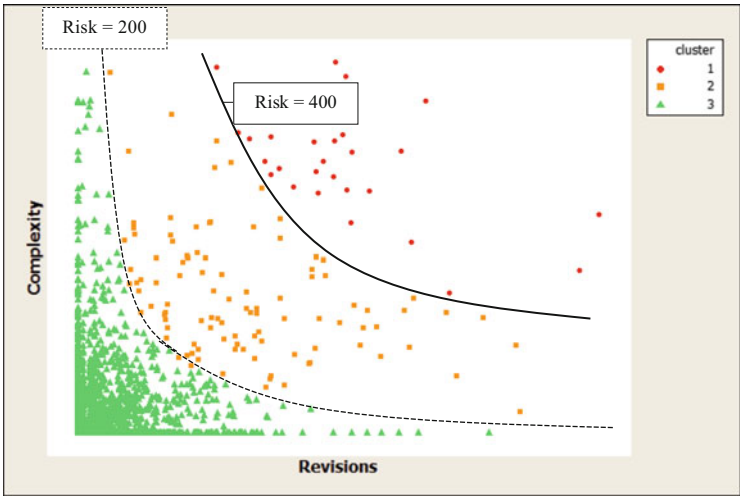


Fig. 14.6 Risk profile of source code files

Fig. 14.7 Information product

14.6 Case Study: Profiling Change Frequency of Models over Time

In this section we discuss a case study of a specific process profiling carried out at Volvo Cars Corporation. The development team of one of the electronic control units at Volvo was concerned with how to profile the changes of product development models over time. The motivation was that if it is possible to visualize how frequently models change over time with respect to continuous development and maintenance, designers can draw conclusions on which models the most development efforts are focused on and if development of one model triggers changes in other models. This information can help designers to understand if they consume their development efforts as it is distributed in their time plan and if the dependencies between models are compliant with the designed architecture. In this case the profiling process described in Sect. 14.4 is much simpler as there is only one property that should be measured—number of changes.

Figure 14.8 visualizes the change frequency of Simulink models over development weeks described in Feldt et al. [7]. Every line in the figure corresponds to a model, while columns are development weeks. The darker spots in the figure are models with more frequent changes within the same week. This kind of profile of changes enables developers to focus on most frequent changes and explore the reasons of them. Several reasons can be behind frequent changes: these can be new functionality development, error corrections, or complex models requiring relatively much time for maintenance. Depending on the reason of changes, the actions are different, such as "no action required," "redesigning unwanted architectural dependencies," etc.

Another benefit of this specific representation is that change patterns between models can be identified. For example, one can observe that every time changing model A after 2 weeks, model B is changed. Intrinsic dependencies that might be among A and B can be identified and managed.

As we see, this kind of profiling does not require explicit establishment of thresholds, but it does not mean that the thresholds do not exist, and no action is required. The figure visualizes models' change frequency which means that there are always a few models that are changed most frequent (darkest ones in the figure). In practice software designers are aware of what changes are expected, and by checking these few models, they can make sure if any unwanted changes have occurred. In case of occurrence, designers can do architectural conformity check and redesign models if necessary.

Change frequency profiles of models are used at Volvo Cars Corporation for systematically monitoring the compliance of developed models with architecture and finding hidden dependences between models.

Fig. 14.8 Change frequency profile of models

14.7 Related Work

Robillard et al. [8] is one of the early studies that attempts to profile the prerelease software product by means of calculating and visualizing all available metrics at the time. They organize the visualization in a compact and simple way so stakeholders with different backgrounds can easily grasp the info. Kitson and Masters [9] investigate possibilities of profiling software processes and categorizing according to their maturity. While software products are becoming more and more complex over years, more sophisticated techniques are required to enable holistic profiling of the performance of prerelease product and processes. Today there are numerous studies that are providing methods and tools for profiling different aspects of organizational performance [10–13]. But before introducing how our research is concerned with profiling, we need to define what is profiling the performance of prerelease product and processes.

Profiling of prerelease product and organization performance can support two types of decisions: (1) related to the economics of software product development [14], referred to as *managerial* in this paper, and (2) decisions related to technology used in product development [15], referred to as *technical* in this paper. Ruhe [15]

recognizes a wider spectrum of decisions in software engineering—e.g., project planning and control, architectural and design decisions, and requirements. However, in the studied organization, it was found that it is easier to discuss metrics and decisions in the chosen two categories without any loss of generalizability while putting stress onto the interplay between decisions and metrics.

Lawler and Kitchenham [16] provided an approach for aggregating measures across organizations and presenting aggregated measures for managers—which is similar to profiling. Although the approach in itself is similar to ISO/IEC 15939 [17], the studied organizations do not use aggregated measures as they do not provide the possibility to quickly guide improvements in the organizations—and in the extreme cases led to measures and indicators that were hard to interpret and backtrack which events caused the indicators to change status, e.g., [11]. Lawler and Kitchenham's approach is similar to the approach used in modern business intelligence tools which aim at providing stakeholders with all available information on request. Although this approach is promising and used in mature disciplines, like mechanical engineering, with established metric systems and theoretically well-grounded measures, the approach has high risks in software development organizations. The risks are related to the potential misinterpretation of data across different projects and products (e.g., even the simplest measures like LOC can be measured in multiple ways).

Organizations starting to use business intelligence tools often face the problem of using these tools in an efficient way after overcoming the initial threshold of establishing the infrastructure for the tools. Elbashir et al. [18] studied the problems of measuring the value that business intelligence tools bring into organizations in a longer run and concluded that these tools are spreading from strategic decision support to support decisions at the operational levels in the company. The value of measures from these tools, according to Elbashir et al., calls for more research. Profiling presented in this chapter supports organizations in effective use of business intelligence.

Balanced scorecards and corporate performance management tools are often considered at top management level as methods and tools for controlling the performance of organization [19–22]. The traces of the balanced scorecard approach were observed at the top management level in our previous work [23]. The studied organization took these measures one step further—making them precise, operational, and automated (in many cases). Profiling helps the organizations in choosing the right measures for each scorecard.

Completeness of information is an important aspect in profiling. It is often a part of the overall information quality and its evaluation. The basis for our research is one of available frameworks for assessing information quality—AIMQ [24]. The framework contains both the attributes of information quality and methods for measuring it and has been successfully applied in industry in the area of data warehousing. In our research we have taken the method one step further and developed a method for automatic and run-time checking of information quality in a narrowed field: measurement systems [25]. In this work we present a method for assessing how complete the information products are; this is a part of

requirements for having high-quality metrics. There exist several alternative (to AIMQ) frameworks for assessing information quality, which we also investigated, for example, Kahn et al. [26], Mayer and Willshire [27], Goodhue [28], and Serrano et al. [29]. The completeness of information is present in all of them in different forms. The AIMQ framework was chosen as it was previously used in our research on information quality—where the information completeness is a part of.

Burkhard et al. [30] found that although the indicators are presented visually, people are surrounded by overwhelming information and miss the big picture. This "bigger picture" in the context of monitoring of software product development means that the stakeholders need to monitor entities that they formally do not manage. For example, project managers monitor projects but also need to understand how the "product has it," for example, what the quality of the developed product is. For stakeholders responsible for parts of product development, that means that they need to understand what the situation "upstream" is—i.e., whether there are any potential problems that might affect their work after a period of time.

Conclusions

Profiling product and organizational performance is concerned with assessing and representing the whole product and process excellence in one picture, where comprising elements of the product are visible in that picture. The method for developing profiles presented in this paper addresses such issues as what the profile should show, which elements should be profiled as building blocks of product, and how the profile of product or process will help in decision making. In this chapter we presented a method which addresses these issues. We presented two industrial experience report on how we developed risk profile of product at Ericsson and change frequency profile of models at Volvo Car Corporation. Both reports are relying on the profiling method presented in this chapter.

The elements that comprise the product are different depending on product characteristics—e.g., source code functions, architectural components, models, requirements specification, etc. At Ericsson, the elements that comprise the product were *source code files* which led to one set of elements in the profile, whereas at Volvo Car Corporation the main elements were *Simulink models* which resulted in a different set of elements in the profile.

The developed risk profiles helped designers to detect the most risky few files out of thousands and refactor them. The change frequency profile helped to detect hidden architectural dependencies and redesign models if necessary.

The next step in our research is to expand the set of available measures to requirements specifications, architecture level metrics, and test metrics. The expansion could provide the possibility to include a wider spectrum of stakeholders in the decision making and analysis of particular profiles.

References

1. Glass, R.L.: Sorting out software complexity. Commun. ACM **45**, 19–21 (2002)
2. Kahn, B.K., Strong, D.M., Wang, R.Y.: Information quality benchmarks: product and service performance. Commun. ACM **45**, 184–192 (2002)
3. Issaverdis, J.: The pursuit of excellence: Benchmarking, accreditation, best practice and auditing. In: The Encyclopedia of Ecotourism, pp. 579–594. CAB International, Oxon (2001)
4. Staron, M., Meding, W., Karlsson, G., Nilsson, C.: Developing measurement systems: an industrial case study. J. Softw. Maint. Evol. Res. Pract. **23**, 89–107 (2010)
5. Staron, M.: Critical role of measures in decision processes: managerial and technical measures in the context of large software development organizations. Inf. Softw. Technol. **54**, 887–899 (2012)
6. Sandberg, A., Pareto, L., Arts, T.: Agile collaborative research: action principles for industry–academia collaboration. IEEE Softw. **28**, 74–83 (2011)
7. Feldt, R., Staron, M., Hult, E., Liljegren, T.: Supporting software decision meetings: Heatmaps for visualising test and code measurements. Presented at the 39th Euromicro conference on software engineering and advanced applications, Santander, 2013
8. Robillard, P.N., Coupal, D., Coallier, F.: Profiling software through the use of metrics. Softw. Pract. Exp. **21**, 507–518 (1991)
9. Kitson, D.H., Masters, S.M.: An analysis of SEI software process assessment. In: Proceedings of the 15th International Conference on Software Engineering, pp. 68–77 (1993)
10. Petersen, K., Wohlin, C.: Software process improvement through the Lean Measurement (SPI-LEAM) method. J. Syst. Softw. **83**, 1275–1287 (2010)
11. Staron, M., Meding, W., Söderqvist, B.: A method for forecasting defect backlog in large streamline software development projects and its industrial evaluation. Inf. Softw. Technol. **52**, 1069–1079 (2010)
12. Wettel, R., Lanza, M.: Visual exploration of large-scale system evolution. In: 15th Working Conference on Reverse Engineering, pp. 219–228 (2008)
13. Voinea, L., Lukkien, J., Telea, A.: Visual assessment of software evolution. Sci. Comput. Program. **65**, 222–248 (2007)
14. Boehm, B.W.: Software engineering economics. IEEE Trans. Softw. Eng. **SE-10**, 4–21 (1984)
15. Ruhe, G.: Software engineering decision support – A new paradigm for learning software organizations. In: Henninger, S., Maurer, F. (eds.) Advances in Learning Software Organizations, vol. 2640, pp. 104–113. Springer, Berlin (2003)
16. Lawler, J., Kitchenham, B.: Measurement modeling technology. IEEE Softw. **20**, 68–75 (2003)
17. International Standard Organization and International Electrotechnical Commission. ISO/IEC 15939 Software Engineering – Software Measurement Process. International Standard Organization/International Electrotechnical Commission, Geneva (2007)
18. Elbashir, M.Z., Collier, P.A., Davern, M.J.: Measuring the effects of business intelligence systems: the relationship between business process and organizational performance. Int. J. Account. Inf. Syst. **9**, 135–153 (2008)
19. Milis, K., Mercken, R.: The use of the balanced scorecard for the evaluation of information and communication technology projects. Int. J. Proj. Manag. **22**, 87–97 (2004)
20. Visser, J.K., Sluiter, E.: Performance measures for a telecommunications company. In: AFRICON Conference, pp. 1–8 (2007)
21. Bourne, M., Franco-Santos, M., Cranfield School of Management. Centre for Business Performance: Corporate Performance Management. SAS Institute, Cary (2004)
22. Wade, D., Recardo, R.J.: Corporate Performance Management: How to Build a Better Organization Through Measurement-Driven Strategic Alignment. Butterworth–Heinemann, Boston (2001)

23. Staron, M.: Critical role of measures in decision processes: managerial and technical measures in the context of large software development organizations. Inf. Softw. Technol. **54**(8), 887–899 (2012)
24. Lee, Y.W., Strong, D.M., Kahn, B.K., Wang, R.Y.: AIMQ: a methodology for information quality assessment. Inf. Manag. **40**, 133–146 (2002)
25. Staron, M., Meding, W.: Ensuring reliability of information provided by measurement systems. In: Software Process and Product Measurement, pp. 1–16. Springer, Berlin (2009)
26. Kahn, B.K., Strong, D.M., Wang, R.Y.: Information quality benchmarks: product and service performance. Commun. ACM **45**, 184–192 (2002)
27. Mayer, D.M., Willshire, M.J.: A data quality engineering framework. In: International Conference on Information Quality, pp. 1–8 (1997)
28. Goodhue, D.L., Thompson, R.L.: Task-technology fit and individual performance. MIS Q. **19**, 213–237 (1995)
29. Serrano, M., Calero, C., Trujillo, J., Lujan-Mora, S., Piattini, M.: Empirical validation of metrics for conceptual models of data warehouses. In: International Conference on Information Systems Engineering CAiSE, pp. 506–520 (2004)
30. Burkhard, R., Spescha, G., Meier, M.: "A-ha!": how to visualize strategies with complementary visualizations. In: Conference on Visualising and Presenting Indicator Systems, pp. 1–9 (2005)

Chapter 15
Industrial Self-Healing Measurement Systems

Miroslaw Staron and Wilhelm Meding

Abstract Automated measurement programs (i.e., placeholders for large number of measurement systems) are an efficient way of collecting, processing, and visualizing measurements in large software development companies. The measurement programs rely both on the software for data collection, analysis, and visualization—measurement systems—and humans for reporting of the data, design, and maintenance of the measurement systems. As the outcome of the measurement program—visualized measurement data—is an important input for decision making in the companies, it needs to be trustworthy and up to date. In this paper we present an experience report on development, deployment, and use of a self-healing measurement systems infrastructure at Ericsson AB. The infrastructure has been in use for a number of years and handles over 4,000 measurement systems in a fully automated way. Monitoring and self-healing of the infrastructure lead to the availability of measurement systems 24/7 and reducing the costs of managing them.

15.1 Introduction

Software metrics provide a foundation for fact-based decisions regarding, for example, software projects, products, and resources. Modern software development organizations utilize them to get insight into the performance of their products or efficiency of the organization. Over time this could lead to companies collecting large amount of data which has to be processed efficiently in order to visualize the data, get an overview, and ultimately support decision making. However, there are two challenges towards efficient use of metrics—lack of standard reusable metrics (base and derived measures) [1] and lack of mechanisms for securing the quality of the information provided to the stakeholders at all times. One of the attempts to

M. Staron (✉)
University of Gothenburg, Gothenburg, Sweden
e-mail: miroslaw.staron@gu.se

W. Meding
Ericsson Software Research, Ericsson AB, Sweden
e-mail: wilhelm.meding@ericsson.com

J. Bosch (ed.), *Continuous Software Engineering*,
DOI 10.1007/978-3-319-11283-1_15

address the first challenge is the ISO/IEC 15939:2007 standard [2] which specifies how measurement processes in software and systems engineering should be structured and how measures and indicators should be defined.

The key elements of the standard is the notion of *measurement system* which is a set of measuring elements assembled together in order to measure a specific quantity. Quantities could vary from application to application (or rather from information need to information need), and examples of these are number of defects in a component, average productivity, and process efficiency. The quantities can either be simple metrics (called base measures in the standard) or more complex ones (called derived measures). A key element in the application of a measurement system is the *stakeholder* who is a person (or a group of persons) who has an *information need*. Stakeholders are roles who need to monitor certain aspects of projects, organizations, or products (addressing their information needs). An example of a stakeholder can be the test manager, whose information need is the test progress in the project (e.g., the ratio between executed and planned test cases). The information need is realized by an *indicator* with associated decision criteria (e.g., progress indicator can notify the test manager about a problematic situation if the execution [base measure] does not meet the expectations [another base measure]). The decision criteria reflect the required values of indicators—e.g., the test progress indicator might have an "unacceptable" level (red) defined when the test execution is too slow (below 80%) and an "acceptable" level (green) when the execution is up to 90% of the plan, leaving the interval 80–90% remaining to be the "warning" level (yellow) of the indicator. The indicator and the associated decision criteria are packaged together with measures into the *information product*.

One of the ways to address the challenge of providing highly reliable measurements is the use of information quality [3] which helps the stakeholder to assess whether the information can be trusted or not. However, this is only an intermediate step towards a more advanced mechanism—a *self-healing measurement system*. Such a measurement system can be defined as a measurement system which can automatically recover from a set of failures, in a manner which is transparent for the stakeholders. In order to achieve self-healing, a number of mechanisms need to be in place as presented in Fig. 15.1.

Each element of the ladder to achieve self-healing is discussed in this paper with particular focus on the top level—self-healing. In this paper the self-healing measurement systems address the following research problem:

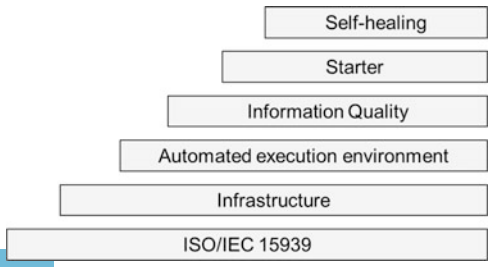


Fig. 15.1 Layers of mechanisms enabling self-healing

How to ascertain continuous delivery of reliable and up-to-date measurement information products?

This research question was posed in order to improve measurement systems by increasing the availability of the measurement data online with high reliability. The research question can be partially addressed through automation of measurement systems, but the continuous delivery and reliability require mechanisms for self-healing.

The paper is structured as follows. Section 15.2 presents the bottom four layers of mechanisms enabling self-healing: ISO/IEC 15939, infrastructure, automated execution environment, and information quality. Section 15.3 presents the top two layers—*starter* and self-healing. Section 15.4 presents the related work, and Sect. 15.5 presents recommendations for other companies. Section 15.6 concludes the paper.

15.2 Mechanisms Foregoing Self-Healing

Delivering measurement information across organizations can be done in multiple ways. The concepts of information radiators [4], metric tools [5], business intelligence [6], or visual analytics [7] were coined for this purpose. The work presented in this standard is compatible with these concepts as self-healing is important for all of them. In order to standardize the discussions and put self-healing in the context, we use the internationally adopted standard for developing measurement programs—ISO/IEC 15939 (Software and Systems Engineering—measurement processes). The definitions of the notion of measurement systems which we use in this paper and in the measurement systems built at Ericsson are taken from ISO/IEC 15939:2007 (Systems and Software Engineering—measurement processes, [2]) and ISO/IEC VIM (Vocabulary in Metrology, [8]). ISO/IEC 15939 has also been adopted by IEEE as IEEE 15939–2007 [9].

15.2.1 ISO/IEC 15939

Measurement systems at Ericsson are defined based on the *measurement information model* defined in the international standard ISO/IEC 15939:2007—outlined at Fig. 15.2. The ISO/IEC 15939:2007 contains a meta-model with the types of measures, which classifies measures into base and derived measures and indicators. This standard is used at Ericsson for defining, designing, implementing, and maintaining measurement systems—e.g., [10].

The ovals in Fig. 15.2 are transitions of the measurement data, while the rectangles are the types of measurement data—different forms of measures and an indicator. This model forms a vocabulary for designing measurement systems.

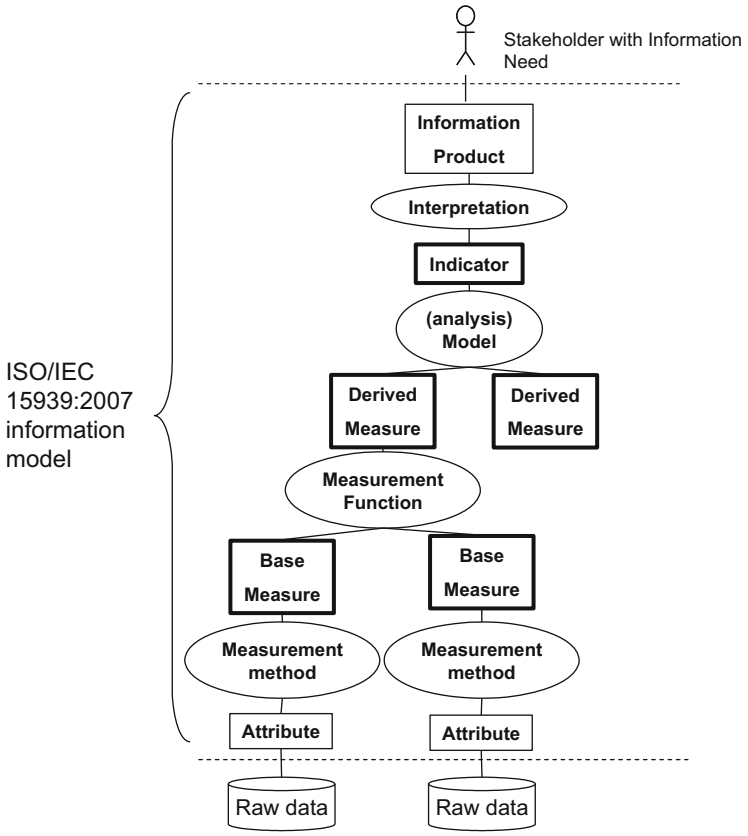


Fig. 15.2 Information model from ISO/IEC 15939, data sources and the stakeholder

A measurement system is a set of measurement instruments assembled in order to measure a quantity of a specified type [8]. The measurement instruments are metric tools which are used to measure a specific entity, for example, a program or a model, and collect a number of metrics from one entity. The measurement system uses the values of metrics from one or several metric tools and calculates indicators from them. These indicators are signals for attracting attention of the stakeholder and are usually calculated from a significant number of data points—values of metrics collected from metric tools. It is not uncommon that in the case of Ericsson, there are over 10,000 data points used to calculate a single indicator.

15.2.1.1 Example of a Measurement System

An example of a design of a measurement system for monitoring test progress is presented in Fig. 15.3—it is a more detailed description of the example presented in the introduction.

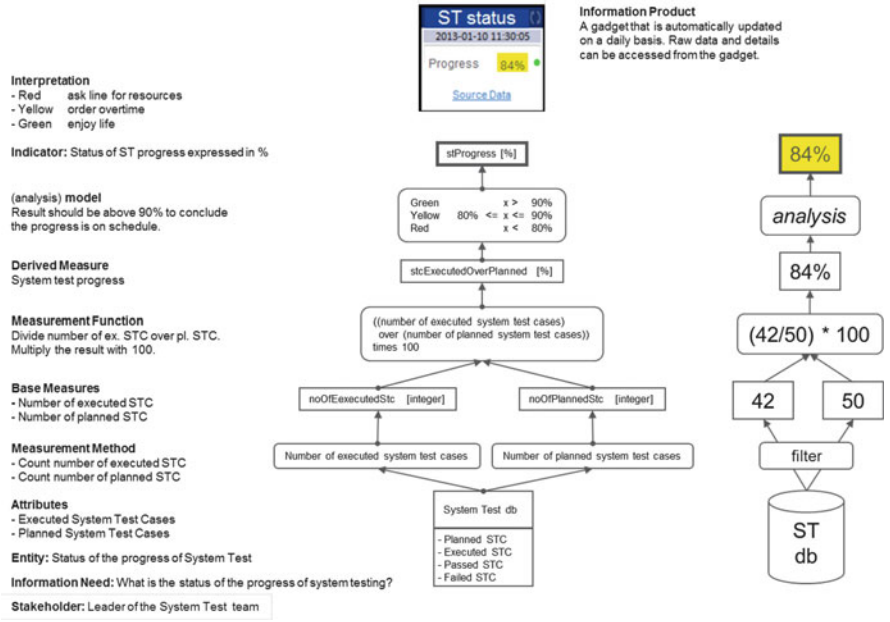


Fig. 15.3 Example instantiation of measurement information model for test progress

The top of the figure shows an MS Sidebar Gadget which is used as a means of visualizing the information to the stakeholders—it is available on their desktop and is updated at regular intervals, e.g., every minute [10].

15.2.2 Infrastructure

Measurement systems at Ericsson are built based on the standard pipes and filters architecture, where the information (values of measures) is processed sequentially. The information flow in measurement systems is presented in Fig. 15.4.

The infrastructure of the measurement systems comprises elements from source files to gadgets, as presented in Fig. 15.4. It defines the structure of files and folders, the databases, and the access interfaces.

The structure of the source files and raw data is standardized based on the organization of the measurement program. It is the structure of folders on the shared network drives and the procedures to store data about product, people, or processes in files or databases. These storage procedures could be version control repositories, defect databases, product performance databases, or central data warehouses common for multiple organizations within Ericsson (e.g., time-reporting database). In addition to the databases, a number of metric tools are used to harvest data directly from products—e.g., [10].

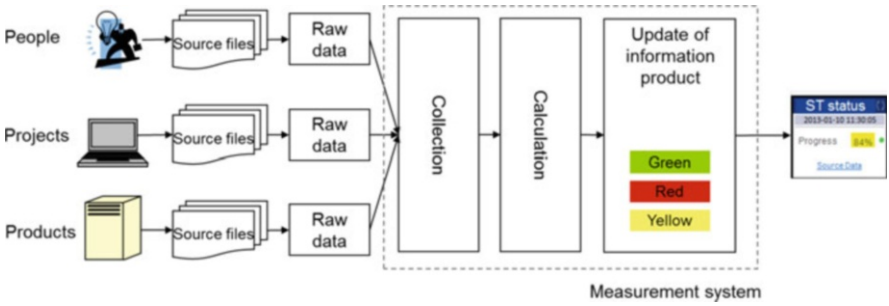


Fig. 15.4 Information flow in measurement systems

15.2.2.1 Visualization of Measurement Products

The measurement systems at Ericsson are implemented using standard tools like MS Excel and MS Sidebar Gadgets for MS Vista or MS Windows 7. This choice of implementation environment has an impact on the visualization of the information products. For example, using D3 library [11] can provide more interactive visualizations than MS Excel. Figure 15.5 shows an example of the same information product visualized as MS Excel and as MS Sidebar Gadget.

As presented in the previous section, this measurement system is updated daily and is available as a gadget which contains a link to the MS Excel file with the details. The visualization—the gadget and the excel file—forms the information product as defined in ISO/IEC 15939.

15.2.3 Automated Execution Environment

The automated execution environment consists of a mechanism to execute the measurement system—at this level no control of the execution process is defined. In the case of Ericsson, this execution environment was *task scheduler* in MS Windows and a list of files tied to it, to execute in MS Excel.

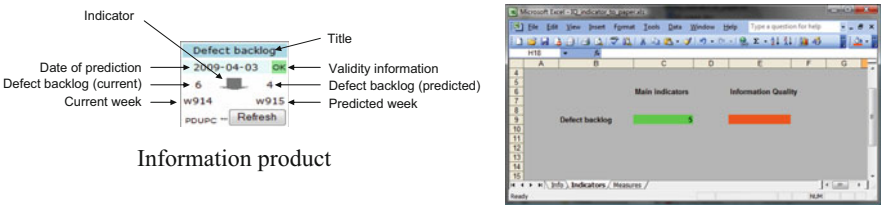


Fig. 15.5 Example of a measurement product—MS Sidebar Gadget (to the *left*) and the MS Excel file with detailed data, structured according to ISO/IEC 15939

In addition to the list of files to execute and the task scheduler, the execution environment consists of:

- Web server to provide the measurement data to MS Sidebar Gadgets.
- Log files storing the information about the execution of the measurement systems.

These elements provide a rudimentary mechanism for spreading the files (information) and monitoring the measurement systems.

15.2.3.1 Measurement Team

The infrastructure and the automated execution environment form a measurement program together with the measurement systems, source files, raw data, databases, and stakeholders. The measurement program is maintained by a measurement team which consists of designers and measurement agents.

The designers are responsible for design, implementation, and maintenance of the measurement systems. The measurement agents are responsible for contacts with stakeholders to elicit information needs in the organization [12] and keep the design of the existing indicators up to date.

15.2.4 Information Quality

Information quality is important to assess whether the information provided by measurement systems can be trusted or not [3]. The annex D of ISO/IEC 15939 with the methods for assessing the quality of measurement systems provides a basic set of criteria. However, we use the AIMQ framework [13] instead, as it provides a quality model of information quality which is more extensive and measurable. The AIMQ framework defines 15 quality attributes of information such as accessibility, completeness, and correctness. In our research we refined this list by identifying two kinds of information quality:

- External quality—how the information is perceived by the stakeholder (semiotics of information: accessibility, appropriate amount, believability, concise representation, consistent representation, ease of operation, interpretability, objectivity, relevancy, reputation, and understandability).
- Internal quality—how the information is obtained and composed (*internals of measurement systems*: timeliness, free of error, completeness, and security).

The external information quality defines the quality of the "design" of the information, e.g., whether a given metric measures what it is supposed to measure. Methods used for empirical metric validation are used to assess the external information quality, e.g., case studies or experiments conducted together with the stakeholders. The following work is particularly useful for this purpose: [14–17].

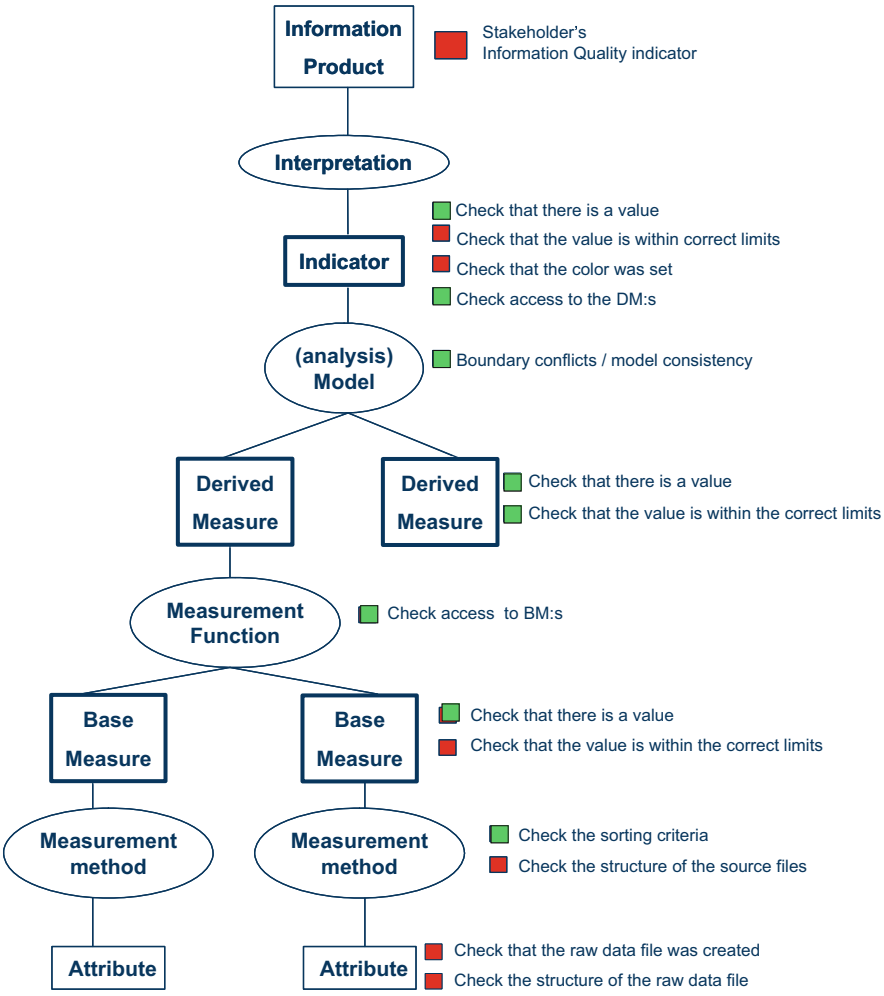


Fig. 15.6 Visualization of information quality checks on the information model (adopted from ISO/IEC 15939:2007)

The questions of the external information quality are handled when building the measurement systems—choosing metrics and indicators—as described in [10]. This is done by the measurement team.

For the purpose of triggering self-healing, however, the internal information quality is more suitable than the external. It defines whether the information is properly calculated and whether it is up to date. This internal information quality can be checked during the run-time operation of measurement systems. Figure 15.6 shows the checkpoints which the information quality algorithms perform to diagnose problems—we check states and transitions. The red rectangles in the figure show examples of problems with the information quality—e.g., checking that the

raw data file was created. The green rectangles show that these steps have no problems.

The checks are independent from each other, and it is enough with only one rectangle to be red to trigger the repair process.

15.3 Self-Healing

In order to address the part of the research question which is concerned with ascertaining the continuous delivery, we used the concept of self-healing from [18] and adapted it to the context of measurement systems. Thus, we define *self-healing* as *the ability of a measurement system to autonomously recover from an abnormal execution*. The self-healing process is conducted in the context of execution environment with multiple measurement systems and addresses the challenges of self-managing systems as described by Kramer and Magee [19].

15.3.1 Self-Healing Process

The self-healing as recovery from the abnormal execution is realized by two elements of the measurement program—*starter.exe* (see Sect. 15.3.2) and each measurement system. This program can also be referred to as the adaptation/healing system in the terminology of De Lemos et al. [20]. The self-healing is organized around three states of the measurement system:

- Normal execution: execution and monitoring of the status of the measurement system.
- Diagnosis: triggered by the error in the measurement system, setting a diagnose what has to be repaired.
- Repairing: repairing the measurement system/s.

As an addition to these three states, the infrastructure logs the statistics on how many times each measurement system was repaired to flag the need for corrective maintenance to the designers of the measurement system—outlined in Fig. 15.7.

This type of self-healing is a special case of the distributed control pattern of self-reconfiguration as presented by Gomaa and Hassan [21] and is based on two main states—execution (which is the normal execution of the measurement systems) and the self-healing (which comprises diagnosis and repair of the measurement system). All states are fully automated and are distributed over the two processes—*starter.exe* and the measurement system.

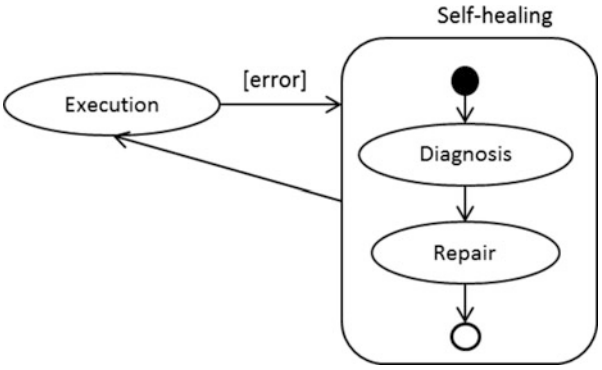


Fig. 15.7 State machine for self-healing measurement systems

15.3.2 Realization at Ericsson: Starter

The placement of the self-healing mechanism in the larger context of the measurement program is illustrated in Fig. 15.8. The measurement systems are a part of the larger context—measurement program—which contains the software which calculated indicators and measures according to ISO/IEC 15939, technical infrastructure, raw data files, and finally the information products which communicate the information to the stakeholders.

The technical infrastructure supporting self-healing is an execution environment in form of MS Windows Enterprise Edition equipped with two main programs—*starter.exe* and *killer.exe* (in the bottom half of the figure). The former is the execution and self-healing software for the measurement systems, and the latter is an auxiliary program to kill a measurement system and recover the whole infrastructure in case of a major failure. Each *measurement system* is realized in MS Excel file with VBA scripts to calculate the indicators according to ISO/IEC 15939,

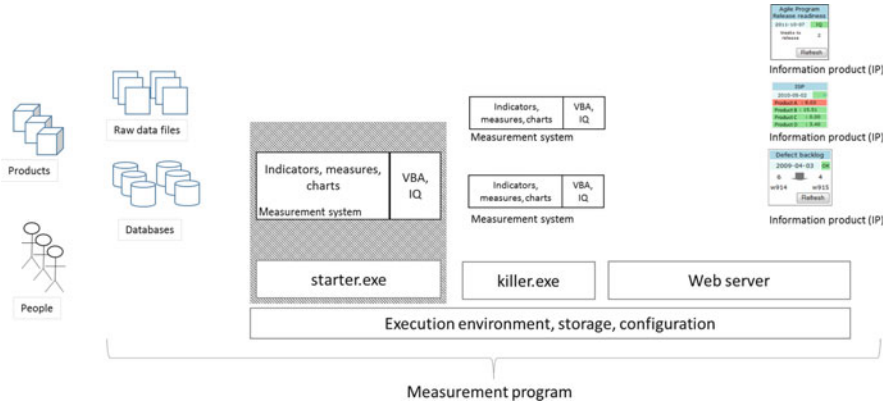


Fig. 15.8 Self-healing (*grayed background*) as a part of the measurement program

to update the information product, and to monitor the information quality of the indicators.

Starter.exe is a stand-alone program implemented in C#, responsible for execution of measurement systems. The execution is done as a multi-threaded execution process. *Starter.exe* monitors the execution time for each measurement system. The algorithm in the starter.exe checks that each measurement system is executed within a given time limit, and if not, then it masks the measurement system for a re-execution. The re-execution is done after all other measurement systems have been executed and is limited to three re-executions. If three consecutive re-executions fail, the self-healing process begins.

Figure 15.7 presents the state machine for the process annotated with how the states are visible to the measurement team.

15.3.3 Triggering of the Self-Healing Process

The process of self-healing of the measurement system is triggered by one of two events—non-availability of the measurement system or deficient information quality [3]. *Starter.exe* also monitors the availability of the web server and through that monitors that the information product from each measurement system is available to the stakeholders [12].

Starter is the program which monitors the execution and triggers the diagnosis process. *Starter.exe* summarizes the current execution status in the form of MS Sidebar Gadget (MS Status in Fig. 15.7) and as a report (accessible by mouse click on the gadget) as presented in Fig. 15.9.

Each line in the report represents a measurement area—one or more measurement systems. The status is represented by the dots and their color—green represents success and red represents failure. The status represents the result of the diagnosis if clicked upon. Different dots represent different checks—execution, update of log files, information quality, and the sum (total). Not all measurement systems have the same diagnostics mechanisms—hence different number of dots.

15.3.4 Diagnosis

After the self-healing has been triggered (i.e., one of the dots becomes red), the first step is to automatically diagnose the problem. Starter.exe is responsible for the process of diagnosis and communicates with the measurement system during the process. There are two parts of the diagnosis process—checking whether the execution of the measurement systems was correct and checking whether the information is correct and up to date.

The mechanism to monitor the quality of the information provided by the measurement systems is based on monitoring a subset of information quality.

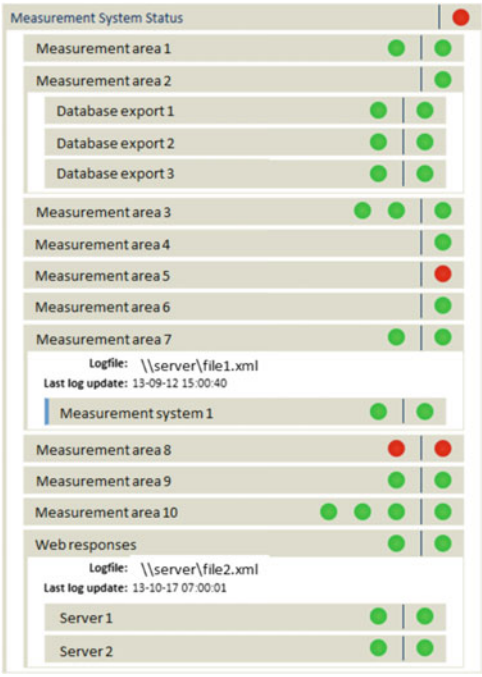


Fig. 15.9 Report from starter.exe with the outcome of the execution of the measurement systems

In the case of Ericsson, we use the AIMQ framework [13] as presented in [3]. The diagnosis concludes with one of the two possible outcomes—the measurement system should be rerun or the measurement system should be repaired.

15.3.5 Repairing

Healing of the measurement system is done in two steps—re-execution of the measurement system and (if the re-execution fails) recovery of the last successfully executed copy of the measurement system.

The re-execution happens if the execution of the measurement system failed, or information quality indicates that there was a problem with the information provided by the measurement system. As there could be numerous reasons for that (see [3]), *starter.exe* re-executes the measurement system and checks whether the execution was successful or not. The process is repeated three times.

If the execution of the measurement system was not correct, *starter.exe* repairs the measurement system by recovering the last validly executed measurement system. This is done by copying either the latest working copy of the measurement system from the archive or by using the original copy of the measurement system (i.e., the first execution copy) from the archive.

After the system has been healed (repaired), the algorithm re-executes the system to test whether the self-healing was successful. If it was not successful, it is re-tried, and if it is unsuccessful, once more the measurement team is notified.

15.4 Scalability

The methods presented in Sect. 15.3 scale to the needs of large organizations—multiple indicators, large data sets of base measures, and a variety of decision criteria. Although the focus of measurement programs should be on the quality of the measures instead of the quantity, the technology behind the measurement programs should allow using any number of measures.

This way of presenting the information is very efficient as it allows to spread measurement systems across the company. Together with the ease of access of information, standard tools and ability to easily combine measures led to a growth of the number of automated measurement systems from ca. 10 in the beginning of 2007 till over 4,000 in 2012. In 2007 the majority of measurement systems were manual, and data collection was costly; in 2012 the majority of measurement systems were automated, and the focus of the organization was on indicators, visualizations, and decision making.

As each MS Excel file contains indicators, based/derived measures, charts, and tables, the number of worksheets which are updated daily is over 35,000. The majority of measurement systems have over ten worksheets and VBA modules, and there are measurement systems which have over 40 worksheets and VBA projects. In total the number of VBA modules in all measurement systems which handle the calculations and updates is ca. 40,000 with ca. four million lines of VBA code. Figure 15.10 presents the statistics over the number of lines of code per measurement system.

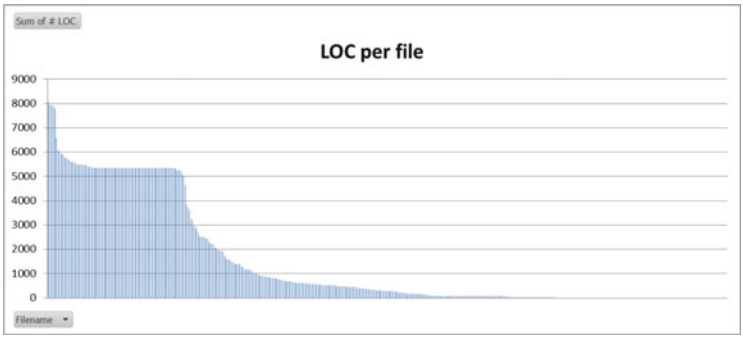


Fig. 15.10 Number of lines of code per measurement system. Each *line* represents one measurement system

Given these relatively large numbers of measurement systems, the company has to use novel methods for controlling the quality of the information in the measurement systems—the self-healing mechanisms. Without these mechanisms, it would be impossible to monitor and control the delivery of measurement information products to stakeholders. Since each measurement system has one stakeholder, we can see how widely the measurement systems are spread across the development unit within the company.

15.4.1 Impact on the Company

The measurement program described in this paper has evolved over a longer period of time. The program started in 2003 with manual measurement processes and data collection to support visualization and decisions about status and progress projects and products [10, 12]. After the introduction of automated measurement systems in 2007, the needs for solid and reliable infrastructure have increased. The needs of the organization for new indicators increased over the time, and the needs for efficiency in handling the data became evident.

The introduction of self-healing mechanisms allowed the metrics team to shift focus from the constant trade-off between maintenance and new development of measurement systems. For the company, which is operating in highly competitive market, such as telecom equipment manufacturers, the need for new insights is constant and must be addressed promptly—the development organization cannot wait for a measurement system to be ready.

On the other hand, maintenance of the existing measurement systems is crucial for having the right insight to make decisions based on the right facts [22]. The introduction of self-healing flipped the coin in the favor of new development—the time spent on solving problems with measurement systems decreased significantly from daily effort to relevant weekly walk-through of problems.

The introduction of self-healing has impacted the stakeholders as well—instead of monitoring the information quality, they could focus on the content of measures as the number of problems visible to them decreased significantly. Without the self-healing mechanisms, each problem could take hours to address (failure, notification, maintenance, re-execution) as all failures required manual intervention of a limited number of human resources. After the introduction of self-healing, the diagnosis and repair processes took minutes, which made it almost invisible for the stakeholders (the "red" information quality status is removed after a few minutes).

The dependency between the measurement systems also made the measurement systems vulnerable to failure propagation—if one measurement system failed, a number of other, dependent measurement systems failed, e.g., refined raw data file. This impacted a number of stakeholders. After the introduction of self-healing mechanisms, the failure propagation is reduced as measurement systems are

repaired and if the failure propagates—so does the self-healing. This leads to potentially non-repairable failures that impact only single stakeholders.

15.5 Recommendations for Other Companies

Based on the experiences with developing self-healing measurement systems, we identified the following best practices for other companies:

- To elevate the metrics competence of the organization, e.g., moving from measuring to addressing information needs, the company should focus on indicators, information needs, and measurement systems—not on metric tools. The relevant indicators provide the organization with the right facts to formulate decisions.
- Use information quality in the initial steps to learn about the most common failures and failure propagations. The knowledge about the propagation of failures is also important for the development of the initial self-healing mechanisms. The danger with starting to develop self-healing without this initial step is that the mechanism gets too complex and handles the "wrong" types of failures—i.e., failures which in practice do not affect the measurement program significantly.
- When deploying the infrastructure, build-in the mechanisms for (simple) self-healing. Once the initial learning threshold has been overcome, the company should focus on introducing the automated mechanisms for handling the most common failures of measurement systems—and in this way move towards a more rigorous self-healing infrastructure.

The above recommendations can help companies to smoothly start with customized self-healing mechanisms for measurement systems.

15.6 Related Work

The approach to self-healing described in this chapter is similar to the component-based self-healing algorithm described by Shin [23]. Shin describes a layered architecture of a self-healing system where the decision about repairing is done in the self-healing layer, while in our case the decision about initiating the self-healing can be triggered by the self-healing mechanisms (starter.exe) or the measurement system itself. The approach presented in this paper is simpler and based on archival measurement systems (components); hence, the testing phase is simplified to a rerun only. The natural extension to this work is self-reconfiguration and graceful degradation [24].

An example of using models when designing software metrics is provided by a recent work of Monperrus et al. [25] where the authors propose a modeling notation

for modeling measures. Although the approach is interesting and model driven (in the sense that it provides possibilities to "model metrics and to define metrics for models" [25]), the approach is not compatible with the ISO/IEC 15939 and regards the resulting measure specification as the final artifact. In our approach we consider the measurement system to be of the core focus in the process, i.e., we take it one step further. A similar approach to modeling of measures is presented by Garcia et al. [26] where models are used to catalogue measures and manage software measurement processes—as opposed to our approach where we focus on generating measurement systems and cataloguing measures in the second place. Garcia et al.'s approach was later extended into a modeling language (similar to ours) which allow modeling arbitrary sets of measures (not only ISO/IEC 15939:2007 compatible) [27].

An alternative to ISO/IEC 15939 method for defining measures was presented by Chirinos et al. [28], which is based on a meta-model for measures proposed by authors created by combining certain aspects of GQM (goal question metric, [29, 30]) into ISO/IEC 15939. One of the reasons for their work was the assumption that neither ISO/IEC 15939 nor GQM has a solid meta-model which can ease the adoption of these approaches. Our work contradicts these results since we show that it is possible and efficient to use a modeling notation directly based on ISO/IEC 15939 and its information model.

In our previous work, we evaluated how much impact the framework for developing the measurement systems had in the organization [10]. The results from the evaluation of the framework showed that it shortened the time required to build a measurement system. The study presented in this paper focuses on how to make the process of developing measurement systems more efficient through introducing graphical notations and automated transformations.

Conclusions

In this chapter we addressed the problem of continuously delivering reliable measurement products. We explored mechanisms for providing self-healing capabilities for measurement systems. We also explored mechanisms that are the basis for achieving self-healing, exemplified on the case of a measurement program at Ericsson.

We presented a study of measurement systems at Ericsson where the mechanisms of self-healing have been applied to provide 24/7 availability of measurement systems for the organization. The mechanisms decreased the cost of maintenance of the measurement program and allowed the company to focus on developing relevant measures, supporting company leadership in their core business areas. It also increased the trust to measurement systems and their use.

The mechanisms of self-healing described in this paper helped the company to increase the responsiveness to problems; many of the failures in

(continued)

measurement systems can be handled automatically, and the human involvement is minimized to only the most severe problems which often require a degree of redesign of the measurement system failing, often due to the changed premises or environment.

The further development of the self-healing measurement systems is to address the problem of changing environment and build mechanisms for self-adaptation of measurement systems. This would allow the measurement program to evolve autonomously over longer periods of time and further decrease the need for human involvement in the maintenance of the measurement program.

References

1. Pfleeger, S.L., Jeffery, R., Curtis, B., Kitchenham, B.: Status report on software measurement. IEEE Softw. **14**(2), 33–43 (1997)
2. International Standard Organization and International Electrotechnical Commission: ISO/IEC 15939 software engineering – software measurement process. In: International Standard Organization/International Electrotechnical Commission, Geneva (2007)
3. Staron, M., Meding, W.: Ensuring reliability of information provided by measurement systems. In: Software Process and Product Measurement, pp. 1–16. Springer, Berlin, Heidelberg (2009)
4. Robinson, H., Sharp, H.: Organisational culture and XP: three case studies. In: Proceedings of Agile Conference, pp. 49–58 (2005)
5. Fenton, N.E., Pfleeger, S.L.: Software Metrics: A Rigorous and Practical Approach, vol. 2. International Thomson Computer Press, London (1996)
6. Williams, S., Williams, N.: Business intelligence readiness: prerequisites for leveraging business intelligence to improve profits. The Profit Impact of Business Intelligence, pp. 44–64. Morgan Kaufmann, San Francisco (2007)
7. Thomas, J.J., Cook, K.A.: A visual analytics agenda. IEEE Comput. Graph. Appl. **26**, 10–13 (2006)
8. International Bureau of Weights and Measures: International vocabulary of basic and general terms in metrology = Vocabulaire international des termes fondamentaux et généraux de métrologie, 2nd edn. International Organization for Standardization, Genève (1993)
9. Association, I.S.: IEEE Std 15939–2007 I.E. Systems and Software Engineering—Measurement Process. IEEE–SA (2007)
10. Staron, M., Meding, W., Nilsson, C.: A framework for developing measurement systems and its industrial evaluation. Inf. Softw. Technol. **51**, 721–737 (2008)
11. Bostock, M., Ogievetsky, V., Heer, J.: D^3 data-driven documents. IEEE Trans. Vis. Comput. Graph. **17**, 2301–2309 (2011)
12. Staron, M., Meding, W., Karlsson, G., Nilsson, C.: Developing measurement systems: an industrial case study. J. Softw. Maint. Evol. Res. Pract. **23**, 89–107 (2010)
13. Lee, Y.W., Strong, D.M., Kahn, B.K., Wang, R.Y.: AIMQ: a methodology for information quality assessment. Inf. Manag. **40**, 133–146 (2002)
14. Bellini, P., Bruno, I., Nesi, P., Rogai, D.: Comparing fault-proneness estimation models. In: Proceedings of 10th IEEE International Conference on Engineering of Complex Computer Systems, (ICECCS 2005), pp. 205–214 (2005)
15. Raffo, D.M., Kellner, M.I.: Empirical analysis in software process simulation modeling. J. Syst. Soft. **53**, 31–41 (2000)

16. Stensrud, E., Foss, T., Kitchenham, B., Myrtveit, I.: An empirical validation of the relationship between the magnitude of relative error and project size. In: IEEE Metrics, 2002, pp. 3–12 (2002)
17. Yuming, Z., Hareton, L.: Empirical analysis of object-oriented design metrics for predicting high and low severity faults. IEEE Trans. Soft. Eng. **32**, 771–789 (2006)
18. Keromytis, A.D.: Characterizing self-healing software systems. In: Proceedings of the Computer Network Security: Fourth International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security, MMM-ACNS 2007, St. Petersburg, September 13–15, 2007, pp. 22–33 (2007)
19. Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. In: Future of Software Engineering, 2007. FOSE'07, pp. 259–268 (2007)
20. De Lemos, R., Giese, H., Müller, H.A., Shaw, M., Andersson, J., Litoiu, M., et al.: Software engineering for self-adaptive systems: a second research roadmap. In: Software Engineering for Self-Adaptive Systems II, pp. 1–32. Springer, Berlin, Heidelberg (2013)
21. Gomaa, H., Hussein, M.: Software reconfiguration patterns for dynamic evolution of software architectures. In: Proceedings of Fourth Working IEEE/IFIP Conference on Software Architecture, 2004. WICSA 2004, Oslo, Norway, pp. 79–88 (2004)
22. Staron, M.: Critical role of measures in decision processes: managerial and technical measures in the context of large software development organizations. Inf. Softw. Technol. (2012)
23. Shin, M.E.: Self-healing components in robust software architecture for concurrent and distributed systems. Sci. Comput. Program. **57**, 27–44 (2005)
24. Shin, M.E., An, J.H.: Self-reconfiguration in self-healing systems. In: Proceedings of the Third IEEE International Workshop on Engineering of Autonomic and Autonomous Systems, 2006. EASe 2006, pp. 89–98 (2006)
25. Monperrus, M., Jezequel, J.-M., Champeau, J., Hoeltzel, B.: A model-driven measurement approach. Presented at the Model Driven Engineering Languages and Systems (MODELS), Tolouse (2008)
26. Garcia, F., Serrano, M., Cruz-Lemus, J., Ruiz, F., Pattini, M., ALARACOS Research Group: Managing software process measurement: a meta-model based approach. Inf. Sci. **177**, 2570–2586 (2007)
27. Mora, B., Garcia, F., Ruiz, F., Piattini, M.: SMML: Software Measurement Modeling Language. Presented at the 8th OOPSLA workshop on domain-specific modeling, 2008
28. Chirinos, L., Losavio, F., Boegh, J.: Characterizing a data model for software measurement. J. Syst. Softw. **74**, 207–226 (2005)
29. van Solingen, R.: The Goal/Question/Metric Approach: A Practical Handguide for Quality Improvement of Software Development. McGraw-Hill (1999)
30. van Solingen, R., Berghout, E.: Integrating goal-oriented measurement in industrial software engineering: industrial experiences with and additions to the Goal/Question/Metric method (GQM). In: 7th International Software Metrics Symposium, 2001, pp. 246–258 (2001)

Part VI
Industry Best Practices and Case Studies

The last part of the book consists of contributions by practitioners at Software Center companies. The chapters report on the experiences at some of the companies while implementing transitions from one step on the Stairway to Heaven to the next step. There are two chapters in this part. The first chapter discusses experiences at Ericsson in implementing agile development in large-scale development of software-intensive systems with very high reliability and uptime requirements. The second chapter is concerned with adopting agile practices and continuous integration in the automotive domain. Due to the nature of the domain, techniques such as modeling and domain-specific languages are used extensively, requiring different agile and testing approaches. Both chapters capture lessons learned and best practices collected and developed in at large, world-class companies and as such are incredibly valuable in terms of industrial validation as well as relevance of the insights.

Chapter 16
Experiences from Implementing Agile Ways of Working in Large-Scale System Development

Jonas Wigander

Abstract Ericsson is operating in a continuously changing business and technological environment. So far, we have as a company been very successful in what we do and are today providing hundreds of different products and solutions to hundreds of different customers.

The prevailing development methodology platform has, up to recently, been our own PROPS project management model combined with various waterfall development methods (e.g., our own methodology for development of the AXE telephony platform, MEDAX).

However, in the last 5 or so years, we have seen a massive transition of our ways of working from waterfall-based methods to Agile (and recently also Lean) development methods.

We will in this story not tell you the details on how we implemented Agile ways of working, but rather share the challenges we have seen moving from one type of development culture (waterfall) to a new, rather different (Agile).

The story is based on a study made by our CTO office, Group Function Technology. The results and conclusions drawn from this study are backed up by other studies made by different Ericsson organizations.

To keep the story short, one can say that the transformation to Agile ways of working was a bottom-up movement from the start. It began in several different product development units independent from each other and with various reasons for why they wanted to change.

As more and more units could show successful results, Agile ways of working became noticed in senior ranks, and finally it was officially exclaimed that we were to become "an Agile company." Thus, the units so far not having implemented Agile ways of working were told they were expected to start doing so.

In these different initial starting points, we find the major reason for why some units gained such a momentum implementing Agile ways of working, and some units still struggle with even leaving the starting blocks; if you have a well-defined objective, your staff will know what is expected and why, whereas if you "are

J. Wigander (✉)
Group Function Technology, Ericsson AB, Stockholm, Sweden
e-mail: jonas.wigander@ericsson.com

J. Bosch (ed.), *Continuous Software Engineering*,
DOI 10.1007/978-3-319-11283-1_16

ordered to," the expected end result is unclear. In the first case, the effect of the change is what matters. In the second case, it is very easy to focus on the exact implementation of different practices, as the end goal is unclear.

After a couple of years of active transition, we now see a pattern in the movement of change. The experiences gained by observing this pattern are what we would like to share with you in this story.

16.1 Introduction

A short, and somewhat simplified, business analysis is needed, in order to understand why the shift of ways of working at Ericsson was seen as very much needed and why Agile methods were chosen.

The telecom business arena changed dramatically in the late 1990s and early 2000s, with new types of operators emerging with new needs and ways of looking at business, often replacing or transforming the traditional PTTs. The traditional telecoms industry rapidly moved towards an IP-influenced technical environment.

As a result, new competitors rose, with new (at the least for the telecoms industry) ground rules and behavior.

Although SW had become increasingly important in our systems over the last decades, Ericsson product development still very much used waterfall methods and processes, built on the knowledge and experiences from HW and system development. Our PROPS project management process had been extremely successful since the mid-1990s, and most of our large-scale system development was organized in massive projects with even more massive total projects keeping the projects together, developing complicated networks.

However, with the changed customer base and new competitors came a more shifting business environment. This was initially handled through a rigorous change management system, consuming much energy from the project management teams.

The complexity of the projects also often resulted in slippage of the internal time schedule, which resulted in ripple effects affecting the projects after the delayed project, with, e.g., delayed entrance of key personnel, test environments, etc. Many development units realized the problem, and new ways of working were introduced, e.g., the Ericsson development pattern "Streamline Development," focusing on managing the changing environment through flexibility and speed in analysis of requirements and decision making.

Two final pieces of the puzzle are still needed to understand the situation that arose in the massive transformation towards an Agile culture in Ericsson's product development: the diversity of technical implementations and the culture in how ways of working are organized.

Even though telecom networks often are very complex, truly being systems of systems, this does not necessarily mean that all the contributing nodes are complex,

or even complicated. There is a rather large diversity in the level of complexity between different nodes but also a large diversity in technologies chosen and distribution of HW vs. SW.

As a result of this, Ericsson has for a long period of time believed in that development units must have the freedom of picking the methodology that supports their business and technology needs. As a company, we have not had one common way of working for many years, other than PROPS.

Ericsson was at that point in time divided into functional areas, e.g., sales, product management, product development, supply, customer integration services, etc. All these functions were tuned together over years of cooperation on providing solutions to the customers, and the main factor keeping the development flow together was the project management model PROPS.

Summarizing the above, when the more massive transformation towards Agile ways of working began, Ericsson's product development was in a situation where it had to manage rapidly changing business needs and requirements in a flexible manner, using a wide diversity of technical solutions, utilizing many different ways of working. Product development is mostly organized in project form, using the project management model PROPS.

16.2 Scaling Agile: A Matter of Change Management

16.2.1 Why Change?

Several different types of reason for introducing Agile development methods and transforming into and Agile culture were identified.

Early adaptors looked at Agile as a vehicle for becoming even more efficient in developing products. One example of this would be a development unit stating "we are rather good at what we do today, and things are going fine; this is the time to consciously look into how we could become even more efficient, as we understand that our business situation is changing." A good quote capturing the need for change even though things are going fine is a consultant talking to an Ericsson development manager: "You guys are brilliant in playing chess; it's a pity though that most of your competitors are starting to play Counterstrike!"

Actively involving their staff, e.g., by discussing what was meant with "efficiency," created a common view of what should be achieved, set objectives accordingly, and then carried on finding and implementing solutions that would meet the objectives set. It should be mentioned that these development units often were smaller, SW centric units. But there are also examples of rather large units doing this as well.

A second category was early followers, units which had problems with, e.g., efficiency. One example would be a development unit with severe quality issues and having ended up with inefficient ways of working. They spent a lot of time

correcting faults and acting on problems, so much that as a matter of fact this specific, very central and important, unit phased the devastating future of having the next release without any business value, as almost all staff and resources were spent on addressing quality issues and system maintenance.

The specific unit exemplifying this category of units then identified the problems they were facing and defined where they needed to be efficiency-wise. They set clear goals on what they expected to fulfill, such as "10 times higher quality," "2 times higher speed in development," or similar (N.B. these are fictional goals, provided as examples, and were complemented with specifications of what was meant with, e.g., "speed"). Then they proceeded by looking inside Ericsson for units that seemed to have managed what they wanted to achieve, but also outside Ericsson for inspiration.

In both the above cases, the commonality was a broad understanding within the development staff of what was the reason for the change and that the objectives set were results of this reason.

But now something interesting happened! As the successful Agile implementations were notice higher up in the hierarchies, it was more or less outspoken that all development units in the largest Business Unit were to adopt the Agile culture and mind-set and implement Agile practices. And in some cases, this was more or less the instruction given: "Off you go, sort it ASAP and report back when you have become Agile!"

And honestly, yes there was support with Agile frameworks, inspirational good practices, lessons learned, etc. But when you don't know *why* you are to "become Agile," it turned out that all this doesn't matter. The problem ended up not in implementing Agile practices such as Scrum, continuous integration, cross-functional development teams, etc. (Ericsson's developers and leaders are excellent technicians; implementing a practice is not a practical problem). Instead, it was clear that in those units who did not have a clear understanding of what they wanted to achieve with transforming to Agile, Agile practices were often implemented by the book, sometimes almost with a religious twist. The implementation then actually became a means in itself and not necessarily a change to solve a specific problem or build a certain capacity or behavior.

In these organizations, it was more common that the transformation was more questioned and challenged, especially when the internal culture was "Why change a winning concept; this way of working has led us to the leading position where we are now!"

16.2.2 Understanding the Greater Picture

As said in the introduction, Ericsson as a company at this point in time was divided into functional areas (e.g., product management, product development, supply, customer integration services, etc.). In this setup, it is very important to understand how all aspects of the development flow work together and that there are numerous

dependencies to cater for. An interesting observation was that as the initial implementation of Agile practices often started as a bottom-up movement, the implementations didn't necessarily cover all aspects of the development flow. This resulted in a mismatch in behavior between different functions, with frustration as a result. Units without a clear objective of the transformation were notably having more problems in succeeding in implementing practices from a holistic view, i.e., to make sure the complete development flow is actually kept together.

When combined with either a lack of understanding of what was to be achieved or a lack of willingness to change, hick-ups and problems inevitably arose, and Agile was blamed as not working for large-scale system development.

Lacking a clear view on what problem one is solving, combined with a focus on the implementation of Agile practices such as Scrum, continuous integration, or continuous analysis, seems to have had an unwanted and unexpected negative effect on the understanding of the greater picture, the "holistic view" of the development system. Ericsson as a company has known how practices add together and how functions and roles interact to produce a well-functioning, efficient development system. Yet, in the situation described above, experiences, knowledge, and old truths were forgotten. This behavior was reported from many organizations, eventually leading to a large work to recapture our experiences on how to develop complicated large-scale systems using large complex organizations.

16.2.3 Proof Points

For several reasons, Ericsson initially had few measurements that give a direction on whether Agile ways of working solved problems or not. Ericsson of course measures a lot of aspects of product development, but these measurements are mostly as locally defined as the ways of working. Thus, there was no common baseline to measure against; we simply did not know what "good" was! For the skeptics, this meant that there was no "solid" proof that the new Agile ways of working were actually performing!

However, there were examples of organizations having had a clear vision on what effects were wanted, which had defined a set of goals supporting the vision, and then started to measure accordingly. Later, several of these measurements have proven that the transformation had started to result in the effects aimed for.

16.2.4 Cross-Functional Cooperation

In some cases, it was difficult for some people who had spent their complete work life working individually to move into the cross-functional mind-set of Agile development. Many remained stuck in the functional silos even though they were organized in XFTs (cross-functional teams). It was not uncommon for people that

have been allowed to have an introvert behavior ("you must understand, they are engineers. . .") having problems all off a sudden and adopting ways of working that require close cooperation with many different persons and roles. A complicating factor here is that some of these persons were highly skilled engineers, with deep competence that we as a company relied on.

Some organizations did not allow different behaviors, whereas others did. Lessons learned are that there is no one easy fix to this problem, but rather the solution seems so far to lie in good leadership and perhaps also to, in some individual cases, allow a pragmatic solution rather than fundamentally force everyone into XFTs.

However, cross-functional does not only mean organizing system engineers, designers, and testers into XFTs. In a large company as Ericsson, cross-functional also means working across functional boarders (see examples above). A clear lesson learned was that if the product management (i.e., the business people) was not actively involved in the transformation, the bigger positive effects would be difficult to achieve. On the contrary, there were also good examples on where there was a close cooperation between product management and product development. It was actually here where the most successful implementations of large-scale Agile system development were found.

The two functional units, HR and finance, were not seen as actively taking part of the transformation and, in some cases, actually were seen as not understanding where product development were moving and what they strived to achieve. This resulted in a disturbing chasm between product development and especially HR, which has taken a lot of time and energy to counteract. The lesson learned here is that when doing such a major change of way of working, completely changing behavior and culture, all functions of the company need to be involved as early as possible. It is easy to blame HR for not having understood the transformation, but they were not necessarily actively included early enough either.

16.2.5 Leadership

Traditional leadership roles have changed or even been challenged as a result of transforming to an Agile culture. The "old" line manager and project managers were two roles that saw a lot of change.

Projects were no longer the prevailing way of organizing the work, so what to do with the project managers who often were skilled and experienced people? In some organizations, this uncertainty led to the unwanted effect that project managers left the organization, or even Ericsson. The lesson learned here was the importance of understanding why the change is happening, the expected results, and that people's skills and competence still are valid. Then let people take an active part of the transformation activities, finding new roles for themselves if possible. But then again, in some cases one has to let go as well. Respect people that do not support the

new culture and ways of working, and try to come to a separation that is beneficial for both parties.

The leadership culture needed to be transformed from the old command and control style to a style more suitable in an Agile culture, a supporting style that builds engagement and motivation and unleashes creativity and innovation. This change was one of the more difficult changes in the transformation for many units, and it has usually taken a lot of focus and energy to build the new leadership culture.

The connection between understanding the reason of the transformation is once again critical, with much more problems in transforming leadership culture in units with low understanding of the objectives, e.g., leaders neither having understood nor bought into the Agile culture. And it was very clear that "command- and control-" oriented leadership still was the prevailing leadership style in several organizations, regardless of the transformation to Agile ways of working or not.

I apologize, but here ends our sharing in this topic. However, we do want to share that leadership culture is key to the successful transformation to any new culture and will need a lot of focus and effort if you are to change the behavior in large development organizations. Not the least is the honest attention of the most senior managers, as people tend to act as they see their leaders *do*, not necessarily as the leaders *say*!

But we will give you one insight though: transparency (i.e., an open climate with open information on status, problems, etc.) is the key to the successful transformation of leadership culture.

16.2.6 Effects on the Development Staff

Many people brought forward that after having implemented Scrum-ish ways of working and executing them for a while, it felt as if the method gave developers means to control and affect their daily life better. Combined with a well-balanced backlog, there was a feeling that one is working with the right things (i.e., effective).

Another positive effect was that some development units managed to involve customer even further in the development, introducing, e.g., demos as part of their ways of working.

The new roles were challenging, not only for leaders but also for developers. The expectations on deep technical competences and broad craftsmanship were sometimes seen as almost overwhelming. There was also (obvious perhaps) a trend towards a need for higher social skills than before, including communication and collaboration.

An interesting observation is this quote from a developer that has moved from a smaller unit with close, rapid customer interaction to a larger organization with higher focus on product quality: "The increased focus on product quality has, among other things, the effect that developers are allowed to take pride in what they have developed, but a negative effect is that there is less flexibility, less of try

and do something new." The initial experience was that teams in which the team members were more stable over time were perceived as more efficient and more fun to work in. This was also true with regard to geographical co-location of team members. Yes, it is possible to have geographically distributed teams. But in terms of efficiency, XFTs were seen as more efficient when co-located.

16.2.7 Thoughts Around "One Size Fits All!" Including Cultural Differences

In Ericsson, we believe that our ways of working must be adapted to the specific challenges the product development organization is operating in, such as technologies, legacy system architecture, customer behavior and expectations, etc. Thus, there will be no "one size fits all" on practice/methodology level. However, the basic foundation such as behavior, principles, and terminology could and should be the same!

But in the case of the earlier implementation of Agile practices, this seemed to be forgotten! Agile development practices were forced upon all units, and in many cases, they were expected to implement a standardized view on practical implementation of practices chosen, regardless of which type of product they were working with. This had a clear negative impact on how Agile ways of working were perceived.

Please note that we are not talking about the Agile culture, which mostly actually was seen as something new and positive.

After a while, the insight that practices of course had to be adjusted to each unit's situation once again was recognized, and the religious discussions have been dramatically played down in most cases.

16.3 Summary

The transformation towards an Agile culture and Agile ways of working (now lately combined with Lean culture and practices) has fundamentally changed Ericsson product development.

How the actual practices have been implemented, and how we have started to manage large-scale Agile system development, is not revolutionary (well, ok in some cases, probably they are). However, the scale of the change has been extraordinary, in terms of the number of units and people involved, but also in the change in culture.

With this chapter, we hope you have appreciated us sharing some of the insights we have gained in managing very large-scale change endeavors.

It all boils down to the old truth: you have to understand what you are leaving and where you are heading and what you aim to achieve.

Chapter 17
Scaling Agile Mechatronics: An Industrial Case Study

Jonn Lantz and Ulf Eliasson

Abstract The automotive industry is currently in a state of rapid change. The traditional mechanical industry has, forced by electronic revolution and global threats of climate change, transformed into a computerized electromechanical industry. A hybrid or electric car of 2013 can have, in the order of 100 electronic control units, running gigabytes of code, working together in a complex network within the car as well as being connected to networks in the world outside. This exponential increase of software has posed new challenges for the R&D organizations. In many cases the commonly used method of requirement engineering towards external suppliers in a waterfall process has shown to be unmanageable. Part of the solution has been to introduce more in-house software development and the new standardized platform for embedded software, AUTOSAR.

During the past few years, Volvo Cars has focused on techniques and processes for continuous integration of embedded software for active safety, body functions, and motor and hybrid technology. The feedback times for ECU system test have decreased from months to, in the best cases, hours.

Domain-specific languages (DSL), for both software and physical models, have been used to great extent when developing in-house embedded software at Volvo Cars. The main reasons are the close connection with mechatronic systems (motors, powertrain, servos, etc.), the advantage of having domain experts (not necessarily software experts) developing control software, and the facilitated reuse of algorithms. Model-driven engineering also provides a method for agile development and early learning in projects where hardware and mechanics usually are available only late. Model-based testing of the software is performed, both as pure simulation (MIL) and in hardware-in-the-loop (HIL) rigs, before it is deployed in real cars. This testing is currently being automated for several rigs, as part of the continuous integration strategy.

The progress is, however, not without challenges. Details of the work split with Tier 1 suppliers, using the young AUTOSAR standard, and the efficiency of AUTOSAR code are still open problems. Another challenge is to manage the complex model framework required for virtual verification when applied on system level and numerous DSLs have to be executed together.

J. Lantz (✉) • U. Eliasson
Volvo Car Group, Gothenburg, Sweden
e-mail: jonn.lantz@volvocars.com; ulf.eliasson@volvocars.com

J. Bosch (ed.), *Continuous Software Engineering*,
DOI 10.1007/978-3-319-11283-1_17

17.1 An Industry in Change

During the last 20 years, the automotive industry has been challenged by considerable changes politically and environmentally due to the contribution to climate change, and technically in line with the embedded software revolution. The result is cleaner and there are more advanced and highly computerized cars. The amount of software in cars grows nearly exponentially with time [5].

As the challenge of computerization is present in other businesses as well, e.g., IT industries, one might ask: what is different with cars? The answer is: at least threefold, considering the market today. First, the expected lifetime of a car, about 15 years, is longer than for most other consumer goods, e.g., mobile phones. This puts different constraints on hardware and software than for products with shorter lifetime. The driver of a car expects to have full IT capability even when the car is a few years old. The second part concerns safety and robustness; the car is a real-time system that is always in a safety-critical situation where human lives are at stake. An airplane in the air usually has time to reboot a malfunctioning system during flight. A few seconds without control is not critical. A car is always on the ground and hence always in risk of collision. The third part is size. As in the airplane the mechatronics is spread over a large vehicle, creating a multi-ECU system with numerous dependent subsystems and functions. Finally, one should not forget the psychology of automotive. We are still steering cars mechanically, although the technology for "drive-by-wire" is available and might even be cheaper. This is since the industry not only has to fulfill legal requirements but also the expectations and opinions of customers. A mechanical steering train feels safe.

The automotive market will continue to evolve towards more computerized mechatronics. If the main challenges currently are zero CO_2 emission and scaling of mechatronic systems, the challenges in the near future will probably involve an enormous increase in connectivity, between cars and between cars and other systems, and autonomous driving. The challenge for any OEM will be adopted.

17.2 The Volvo Way

At present day, R&D at VCG is an interesting mixture of traditional automotive requirement engineering using waterfall strategy and agile in-house development. The in-house initiatives started in groups responsible for product features dominated by software functionality, such as active safety, motor control systems, body electronics, and hybrid technology. Nevertheless, the majority of systems in the car are still developed using a waterfall process, with requirement engineering followed by outsourced development of components and software. Then the ECU is finally delivered, whereas physical integration, rig, and finally system level testing can be conducted at the OEM. One should also note that the overall communication architecture is still developed using waterfall practices

[6]. Hence, we are considering agile development inside ECUs only, which often translates to creation of agile feature or function groups. The ECU is usually tailor-made for its domain, which gives the associated teams a partial freedom to choose their own strategies. Only when the interface towards other ECUs needs to be updated that the development turns non-agile. This distributed process evolution is ongoing, and it is unwise to speculate in future directions, although details will be discussed below.

The new standard for automotive software architecture, AUTOSAR [9], has improved and facilitated distributed development in many ways, although the standard is still not 100% developed and comes with some teething problems. AUTOSAR defines applications (compared with, e.g., mobile applications) for embedded ECUs, introducing some degree of platform independence. An AUTOSAR application need not be ECU specific, and its format is the same even for very different ECU platforms, hence, facilitating post-deployment and standardized updates of software. It is also important that AUTOSAR helps suppliers to develop standardized solutions for common tasks, as IO, diagnostics, and communication. Finally, AUTOSAR defines a standard for integrating applications and eventually also configuring the ECU services, which can be automated. This is extremely powerful since it opens a door to continuous integration, test, and deployment.

17.3 Agile Domain-Driven Development

The development of in-house software at VCG involves several different platforms and external suppliers. The associated physical systems and requirements are also very different. Hence, the development of software in an HMI system is very different from that on a brake ECU, which in turn is different from developing the software controlling the coupe climate system. This demonstrates the main challenge of developing a complex optimized real-time system distributed over multiple ECUs connected in a complex network. Not only the hardware and ECU platforms are different but also the domains.

A car platform is the base of the vehicle, electronically and mechanically. On top of the platform, different car products are built. Significant parts of the software, hardware, and mechanical construction are reused among the products, and the number of software variants can hence be minimized. A new platform is developed as a project, spanning over several years. The present product platform at VCG is currently in the final stages of development, and the first products are being finalized. This is also the first platform where considerable amounts of software in several ECU systems are developed in-house.

Since the new platform is developed as one single project, almost all systems are developed or redeveloped in parallel. There is indeed reuse of knowledge and solutions from previous platforms, but only while the same supplier is kept when switching platform. Otherwise, the knowledge will be limited by the documentation

stored at VCG, in the form of requirements. The most striking consequence of this parallel development is the late learning associated with the waterfall process. Faulty assumptions and bugs are discovered at the first real integration [10], leading to numerous task force initiatives and enormous workload near the project deadline. Consequently, complexity and its associated problems drive VCG towards more in-house software development.

Changing to in-house software development is however nontrivial. The car involves multiple and quite different ECU platforms, although AUTOSAR has initiated a process of standardization. Moreover, each ECU platform typically has an IO connecting it to sensors and actuators. Consequently, an overall R&D decision has been to restrict in-house development to feature or controller applications (following the AUTOSAR definition [1]), leaving the platform software and IO to the Tier 1 supplier. This work split will also give the in-house developers access to the car network, which is designed by VCG.

However, as associated mechanics, hardware, and software are developed in parallel for each ECU subsystem, there is still a substantial part of the development being outsourced, based on requirements, also for the in-house ECU. In some cases the mechatronics is outsourced to other suppliers than the ECU platform, which is demonstrated in Fig. 17.1. It is common to have these three parallel development lines for each function, which are merged late in the project. Thus, there is still a significant risk of problems caused by late learning. The first integration of software in an ECU, or the integration of the ECU itself in the mechatronic system, is then an extremely important event during the development.

One should note that more thoroughly, requirement development or analysis upfront might not solve the issues associated with this waterfall process. This is since it is extremely difficult to foresee details of a solution before it is developed, which is also the main reason for agile development overall. The VCG solution, which is popular in automotive industries, is a combination of methods where we distinguish between domain-driven development (DDD) and model-driven engineering (MDE). The idea of MDE is to develop models of not yet existing, not delivered, or just unmanageable physical parts. These models can be combined with models of the control software, creating an executable model of the complete subsystem. Development and verification of embedded software can then be conducted on the developers' regular PCs. DDD is a method to involve domain experts directly in software development. Domain-specific languages (DSL) can be specialized for different domains and provide tailor-made abstraction also of rather advanced code. The gain is speed and considerable simplified reuse of easy-to-read models. Software development by domain experts through DDD is an important enabler for agile development, as short loops are facilitated. MDE will boost this process even further in mechatronic subsystems, where plant models can be used for reliable virtual verification. Early learning is thus possible (Fig. 17.2). Finally, it is believed at VCG that software development, testing, and documentation will be most efficient if the same tool or language is used for all three purposes. The motivation is simply the minimized tool chain, which is easier to learn. Thus, we are

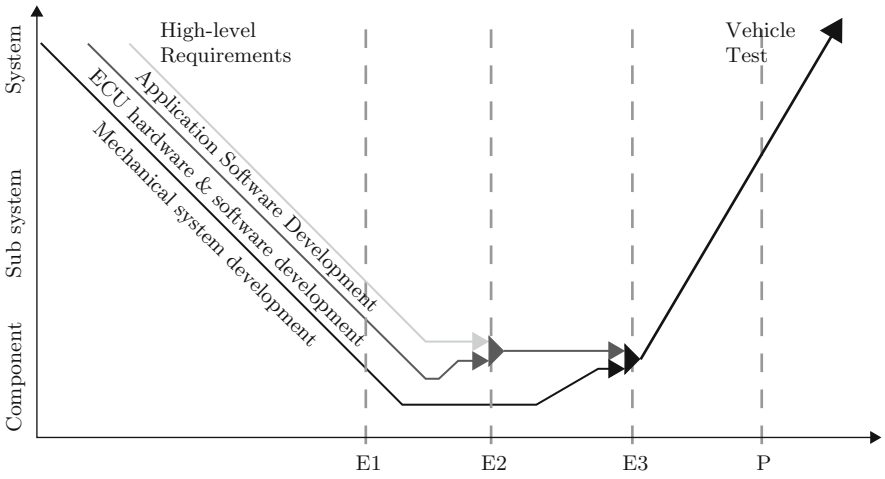


Fig. 17.1 The V-model as it looks at VCG for a car development project. Software, hardware, and mechanical development happens in parallel and is integrated at certain points during the project

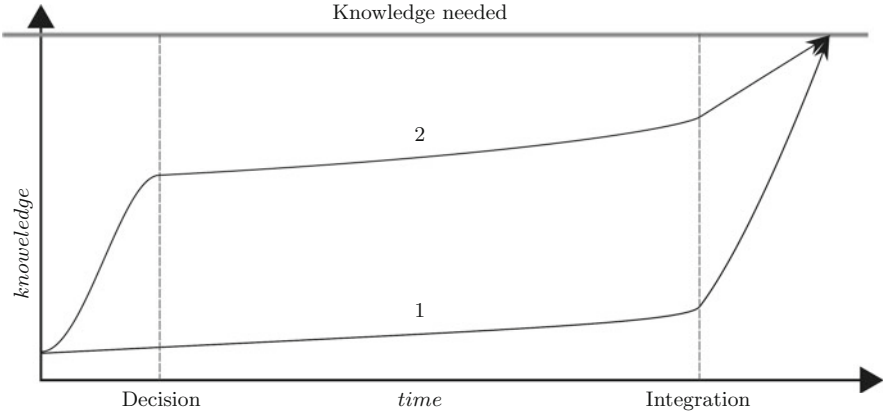


Fig. 17.2 Using DDD and MDE techniques, it is possible to boost early learning although real integration is not yet possible. (1) Demonstrates the typical knowledge curve for waterfall development using requirement engineering, where no learning is possible from simulations. (2) Demonstrates the corresponding curve using virtual verification and plant models. Much is learned using the models, although some faulty assumptions may still cause some late learning

currently trying to integrate as much as possible in the Simulink platform used by the developers to develop AUTOSAR applications.

Obviously, while trying to predict or create models of parts, communication, or IO which are not yet realized, assumptions are made [7], of which several will be faulty. Nevertheless, there is a strong consensus and research results [8] suggesting that the overall speed will benefit from this approach although the risk of anomalies

due to faulty assumptions is always present. The risk of finding serious bugs in late integration, without previous modeling, is too large.

As described above, the methods differ between different sections of R&D. Nevertheless, the overall process is roughly the same. A common system model manages the complete architecture with all its networks, ECUs, and application software components. The system model provides the ECU with interfaces, whereas agile development can be conducted at each ECU on a more flexible internal architecture. Changes in the inter-ECU communication are, however, still rather slow due to the complexity and manual configuration of the network.

The interfaces of AUTOSAR applications are defined using an AUTOSAR meta-model and can be stored in the main system model. These AUTOSAR models are described and exchanged as XML. This AUTOSAR XML is the main format for communicating architectural data, both between groups at VCG and with suppliers. VCG is presently using AUTOSAR 4.0.3 [9].

The use of a system model and DSLs forces VCG to maintain several model transformations. Efficient and automated model transformations are generally believed as a key to efficient DDD or MDE. But they are also considered as the main bottleneck. Automation and robustness are essential. At VCG the AUTOSAR XML, being the main exchange format part from C-code, is involved in all model transformations, communicating the architectural information about the system. Examples of these automated transformations are as follows: code generation, which is the creation and maintenance of the interfaces of AUTOSAR application in the form of Simulink models [2], and virtual integration, in which larger Simulink models representing complete ECUs or even larger subsystems are generated from application models and architecture XML. It is important to notice that all these transformations are developed in-house, although big parts are also bought off the shelf, as the C-code generation.

Before a change or a new function can be deployed in a vehicle, it has to pass three levels of testing: the developers unit test, MIL test, and HIL test. The HIL testing, where the ECU with controller software is running in a model environment, is widely used. This is since it represents the first chance to test Tier 1 software, eventually in combination with in-house software. The MIL test is not yet fully utilized at R&D, but state of the art is to generate Simulink ECU models which are tested against plant models (mainly physical and environmental models). These MIL tests can also be extended to full-scale virtual cars running complete drive cycles. Comparing MIL and HIL, the latter is important for configuration, integration, and performance test. Detailed temporal testing is however difficult as the HIL rig runs in real time. The MIL test is a white box test technique, offering full insight in the system model while running. This is a powerful technique to test temporal functions and for edge testing, although it requires large computers and tedious maintenance.

17.4 The Art of Virtual Verification

The idea of virtual verification at VCG is to find functional and communication errors early, preferably in a fully controlled white box MIL environment. Regression tests on virtual cars can detect issues on system level, without expensive rigs or prototypes. Moreover, it is extremely useful for edge testing of potentially dangerous maneuvers as for tedious (simulated) long time wear and tear test. Continuous integration in a virtual verification environment can provide very fast and reliable feedback, although the full system (car) is far from ready.

The final tests before deployment in cars are done in HIL rigs and in rigs combining simulated environment with real devices (e.g. engines, lamps or other mechatronics). Regression tests are automated here as well, although the feedback time is generally longer. State of the art for HIL test is feedback within 24 h. Thus, using the existing framework and avoiding changes in the inter-ECU communication, we can in principle deploy a new software version in a test car within 24 h.

Considering the in-house teams at VCG which are active in virtual verification development, we note that domain-specific languages (DSL) in physical domains tend to be very specialized. Since several domains (both physical and software) are modeled, tools and languages can be very different. DSLs facilitate modeling and enable reuse and readability but make integration and model transformations (as code generation) trickier. Since modules in different languages also have different interfaces, or even interface semantics, it is difficult to agree about common interfaces. The same holds for exchange and integration formats. A common conclusion today is that C-code is the natural model exchange format for executable models. All modern tools for software modeling and the leading tools for physical modeling are shipped with C-code generation capability. The challenge is however to communicate interfaces and variable formats from different tools and domains, as well as to integrate different paradigms of modeling in a super model which is executable. Today at VCG, executable full vehicle models are constructed using subsystem models from different domains delivered as Simulink models or as generated C-code which is manually harnessed in Simulink. It is a tedious work to ensure that all parts of this super model compile and run correctly together.

An important challenge which one soon realizes after integrating a few different physical or software models in a virtual verification environment is the lack of good architecture tools. In the coming years, much effort will be spent both on constructing repositories for various plant models and on architecture solutions. The architecture challenge is complicated even more since different plant models of the same part or device are used for different purposes. Moreover, the plant model of a device may also be replaced by a real device, using a rig, while other parts of the system remains virtual. There are numerous combinations, and in all cases the architecture has to describe both the software system and the mechanical or electrical system such that the system model is executable.

17.5 Continuous Integration at VCG

During the past 2 years and the startup of the research-industry collaboration Software Center, continuous integration initiatives have been initiated at several in-house software groups at VCG. Although the R&D as a whole still defines itself as a traditional waterfall process requirement engineering organization, several groups, driven by the need for speed in software innovation-oriented areas, have successfully implemented local agile processes and ad hoc tool chains. Hence, the common consensus is that the single ECU can be agile [6], although changes involving other ECUs or supplier technology are still required to follow the waterfall workflow. The majority of the car software is still developed by Tier 1 contractors. Nevertheless, the trend during the past years has been an ambitious increase of the in-house software, with all the challenges that follow in the highly distributed mechatronic vehicles.

Considering the agile ECU the main challenge for continuous integration has been to adopt the tool chain used for domain-driven development for automation. Several tools as well as the newborn standard AUTOSAR 4.0.3 are immature and require numerous workarounds and hacks. Problems occur because of varying interpretations of the AUTOSAR standard. This forces the integration teams to design scripts for automated translation of the AUTOSAR exchange documents between different interpretations. The maintenance and quality assurance of these scripts are considered to be great challenges. Another challenge is the incorporation of DDD transformations in the AUTOSAR tool chain and the automation of software builds. It has been shown that the DDD+MDE workflow is very efficient at VCG regarding speed and reuse in in-house software development, as it provides tools both for directly involving domain experts in the software development and since it facilitates simulation of, e.g., mechatronic subsystems before the real hardware and mechanics is present [8]. The latter enables early unveiling of faulty assumptions regarding mechanics and the surrounding system, assumptions which in a waterfall process would have been discovered first at the first real integration.

Continuous integration is currently defined at VCG as automated file collection and build of ECU software followed by regression tests in HIL rig and in some cases MIL environment (running a model of the ECU in a virtual environment). The HIL tests require nightly testing to complete. The feedback time is 24 h at best. One should note that some ad hoc reconfiguration is usually needed to integrate new versions of the supplier part of the software in an ECU, or if the ECU interface towards the cars network is updated. Nevertheless, automated builds with feedback on ECU level can be executed in about an hour, without the time-consuming regression test execution.

The development towards continuous integration at different groups has followed the outsourced development of the used platform. Hence, true integration has not been possible until real ECU hardware has been available. Moreover, an agreement with the ECU supplier has to be made, establishing a common build

environment (including the automated RTE generation and other support services in AUTOSAR [1]).

Agile development does not follow the waterfall model. Obviously, architecture, applications, and models are always under development. This is why continuous integration, virtual or real, is so essential for agile development. In an agile process at VCG, the requirement, which was earlier used to order Tier 1 software, may now be written after the software has been implemented. The traceability (required by the automotive ISO 26262 standard [4]) between requirement, model, test, and code artifacts must however be complete in the end.

17.6 Process and Organizational Aspects of Agile

The most striking change, apart from the increased speed, when a group used to requirement engineering moves on and start developing in-house software is the natural breakdown of the waterfall process, which is translated into an agile process, as far as the tools and the organization allows (Fig. 17.3).

In a waterfall process, the key is complete and the requirements and specifications are well written. Another key is to have long development cycles, such that faulty assumptions and diverging design decisions among the developing parties, e.g., due to incomplete specifications, can be redeveloped in time. This works as long as the systems involved are not too complicated and possible to design without reliable testing and while the market allows the long development cycles required.

The most common reason to switch to in-house development is not time to market, but frustration over slow progress and high prices when updates are required. Costly updates of an existing outsourced software which is not working as expected are a common and strong driver.

When the decision is made and the group finally has built up the competence for in-house development and stable ECU builds, the process and workflow utilized by the development teams usually change towards more agile methods. There are examples of formation of "supplier groups," i.e., keeping the waterfall workflow with requirement design followed by handover to software developers, but the most common way is to use existing groups and introduce ad hoc collaboration and "function teams." Since the groups at VCG responsible for functions or features usually are cross-functional (i.e., involves hardware, software, and mechanics) and since testing groups and facilities are usually located nearby, there is a natural formation of "cross-functional" development teams, or at least closely collaborating individuals. When an integration team is finally established and the ECU supports in-house integration (usually require supplier negotiations), continuous integration (virtual or real) and an agile process will be feasible.

One should note that agile development is and will be difficult as long as the outsourced development of the hardware is parallel with the software development

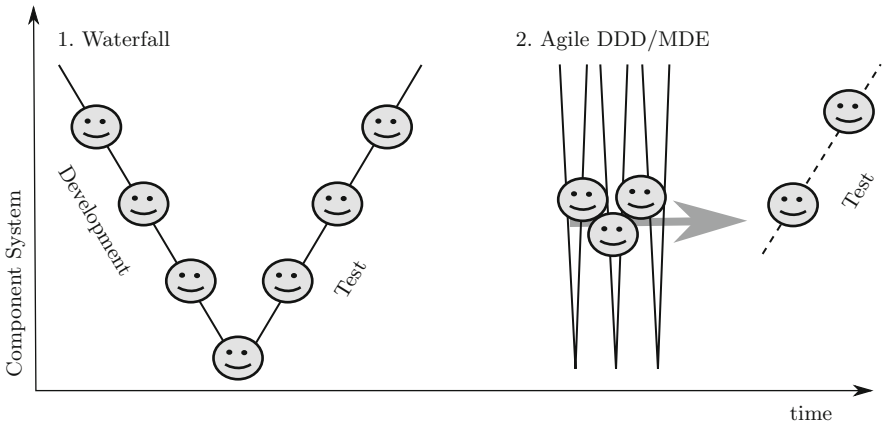


Fig. 17.3 Illustration of the changed workflow when moving to an agile process. While the waterfall process (1) consists of successive handovers, a function team using agile methods (2), DDD and perhaps MDE, can conduct fast successive loops spanning from system level to component development. The agile process is finalized by the "standard" handovers of solutions for final tests on the subsystem (rig) and system (car) levels

in the same project. This parallel development can be facilitated and early learning can be boosted using MDE [8]. Nevertheless, it is clear that improved planning of hardware and software projects could improve the software development.

An important aspect of the in-house development is the construction of development teams and roles. Considering the complete needs of architecture, software development (i.e., software modeling by domain experts), requirement development, and integration/automation, different groups at R&D have chosen slightly different strategies. It is common to assign one integration team, managing tool chains, integration, and "secondary software" (required for model transformations, code generation, integration configuration, etc.). The idea is to let domain experts focus on modeling the domain software and not bother with the usually much more code-oriented tools of integration and build environments. Nevertheless, at least one large group has intentionally distributed the integration responsibility among some of the developers. This will require larger investments in increased competence but will lower the risk of "dependence of the few," e.g., when integration falls when the experts are gone. The complete integration team is still located in the same group.

Attempts have also been made to outsource the development of the secondary software. Nevertheless, it appears at least from initial studies that this is hazardous. In an organization utilizing DSLs for agile development, the secondary software will be in constant change, driven by tool version updates, new features, new agreements, etc. Thus, the development of the application software and the secondary software has to be equally agile.

At all in-house developing groups, there has been a natural grouping of developers contributing to the secondary software in the integration teams, mainly since

the software engineering competence required for integration is similar to that of model transformations and code generation. The formation of integration teams also follows the physical car architecture, as does the R&D organization. One ECU is managed by one integration team. The only exception is the "Electric Propulsion System" Department, where one integration team supports four ECUs. These ECUs are, however, rather small. As a consequence it is straightforward to move or reuse software components between EPS ECUs, but difficult to move an application from active safety to the ECU running the body functionality. This is since modeling strategies and integration environments differ between different integration teams. Nevertheless, this is currently not a problem. Usually there is no point in moving around applications, since the ECUs are domain oriented. Having different Tier1 suppliers also makes reuse or relocalization difficult. Hence, we see an organization reflecting the mechanical and functional architecture.

It is natural to argue that centralized, larger ECUs collecting the vast majority of the in-house software then would save integration personnel and money. The case is, however, not that simple, which is discussed in the next section.

17.7 Optimizing Automotive Mechatronics

Judging from the previous sections, some readers might find the car architecture with the order of 100 ECUs overcomplicated. Why not facilitate software development by replacing several ECUs by one and adopt a standardized and easy-to-maintain tool chain? The answer is, however, nontrivial although it may be worthwhile in some cases.

The cost of producing a vehicle is still dominated by the vehicle production. The R&D cost is hence relatively small and the software share of this R&D cost is generally believed to be less than 1/3, although this number increases rapidly. Exact numbers for the software-related parts of the development cost are however difficult to find, and no numbers will be mentioned here. Nevertheless, the software part of the development cost is increasing for modern vehicles and will continue to increase. Moreover, the cost of recalled vehicles due to software bugs can be enormous [3]. One could also argue that the slow pace of software development, at system level, which is implied by the complex network and mechanical optimization may not work in a future more software feature-driven car market.

The common understanding in the automotive business is, correct or not, that the bulk of vehicle cost is hardware and mechanics. This has several important implications. First, standardization of automotive ECUs is held back, with very few exceptions. It is almost always considered as worth the effort to switch to a lighter or cheaper platform. It is not unusual that platforms are built from scratch. Second, the cabling represents an important contribution to the total vehicle cost. Hence, centralization of functionality in a mechatronic system may not be profitable, although the development would be facilitated a lot. Numerous ECUs in the car have extensive and sensitive cabling to sensors and actuators. Moreover, it is not

uncommon that timing requirements on the functionalities implemented in these ECUs requires the software to run locally. Motor controllers are a good example. If key customer functionality is included, it cannot be centralized and the development will have to struggle with the disadvantages of nonstandard platforms.

The AUTOSAR initiative has raised the issues of reuse and portability of software. However, these needs are not yet visible in the industry, mainly since the organization of automotive OEMs as VCG is reflecting the ECU architecture. However, when the power and markets of post-delivery updates of car software are recognized in large scale, the situation may be different.

References

1. AUTOSAR. http://autosar.org/
2. AUTOSAR support in MATLAB and simulink – automotive industry standards – MathWorks nordic. http://www.mathworks.se/automotive/standards/autosar.html
3. Toyota recalling 1.9 million prius cars. http://www.usatoday.com/story/money/cars/2014/02/12/toyota-prius-recall/5414055/
4. ISO/DIS 26262 road vehicles – functional safety. Tech. rep. (2011)
5. Ebert, C., Jones, C.: Embedded software: Facts, figures, and future. IEEE Comput. **42**(4), 42–52 (2009)
6. Eklund, U., Bosch, J.: Archetypical approaches of fast software development and slow embedded projects. In: 2013 39th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), Sep 2013, pp. 276–283
7. Eliasson, U., Burden, H.A.: Extending agile practices in automotive MDE. In: XM 2013-Extreme Modeling Workshop, p. 11 (2013). http://ceur-ws.org/Vol-1089/proceedings.pdf#page=19
8. Eliasson, U., Heldal, R., Lantz, J., Berger, C.: Agile model-driven engineering in mechatronic systems – an industrial case study (2014), Models 2014 ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems
9. Fürst, S., Mössinger, J., Bunzel, S., Weber, T., Kirschke-Biller, F., Heitkämper, P., Kinkelin, G., Nishikawa, K., Lange, K.: AUTOSAR–A worldwide standard is on the road. In: 14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden (2009). http://www.win.tue.nl/~mvdbrand/courses/sse/0809/papers/AUTOSAR.pdf
10. Mellegard, N., Staron, M., Torner, F.: A light-weight defect classification scheme for embedded automotive software and its initial evaluation. In: 2012 I.E. 23rd International Symposium on Software Reliability Engineering (ISSRE), Nov 2012, pp. 261–270

Index

A
A/B testing, 4, 157
Academic excellence, 9
Active safety, 212
Agile development, 15–26
Agile development practices, 6
Agile evolution, 83–92
Agile organizations, 39–49
Agile software development (ASD), 6, 39
Agility, 40
Alternative implementation, 161–162
Architects, 11, 39–49
Architectural rules, 81
Architecture
  documentation, 44
  education, 47
  evolution and refactoring, 40
  runway, 40
Artifacts per potential executions (APPE), 113
"Automated build", 109
Automated measurement, 183
Automated testing, 129–131
Automated test suite, 15
Automotive industry, 211
Automotive mechatronics, 221–222
Autonomous box parking, 118–122
AUTOSAR, 216
Axis Communications, 9

B
Backlog building development, 90
BAPO model, 18
"Big-bang" integration, 8
Big data, 5
Burndown charts, 107
Business, 16
  ecosystem, 15
  goals, 51–65

C
CAFFEA framework, 39
Chalmers University of Technology, 9
Change frequency, 177–178
Chief architect, 39
Civil security, 20
CIViT, 97–105
Co-creation, 16
Code reviews, 46
Collaboration, 29–36
Collaborative research, 35
Communications networks, 21
Configuration managers, 20
Consumer electronics, 143
Continuous assessment, 168
Continuous deployment, 8, 15–26
Continuous innovative development, 90
Continuous integration, 8, 18, 97–105
Continuous integration environment, 7
Continuous software engineering, 3–13
Cross-functional, 6
Cross-functional development teams, 17, 206
Cross-functional teams (XFTs), 86, 207
Customer
  collaboration, 16, 83
  feedback, 16, 145
  first-features, 85
  relationship management, 85
  requests, 84

J. Bosch (ed.), *Continuous Software Engineering*,
DOI 10.1007/978-3-319-11283-1

Customer (*cont.*)
responsiveness, 155
satisfaction, 88
support, 85
unique-features, 85
Customer-specific teams, 83–92
Cyber-physical systems, 125

D
Data analysis, 29
Data collection method, 20, 29
Data-driven development, 155
Data-driven software development, 155–163
Data sense-making, 29
1st Deployment Speed, 59
Diagnosis, 193–194
Diagnostic data, 150
Directed Acyclic Graph, 109
Distributed teams, 58
Domain driven development (DDD), 214
Domain specific languages (DSL), 211

E
Ecosystem, 16
architecture, 19
management, 15
organizing, 19
strategy, 19
Efficiency, 16
Engine control unit (ECU), 218
Ericsson, 9, 84, 203
Erosion, 46–47
ESAO model, 16
Evolution, 49
Evolution speed, 60
Exploratory tests, 129

F
Feature, 161
Feature backlog, 160–161
Feature-boxed development, 90
Feature dependencies, 62
Feature experiments, 17
Feature usage, 152
Feedback loops, 16, 83
Flexibility, 16
Force protection, 20
Forums, 44

G
Gap analysis, 161
Governance architect, 39
Graphical user interface (GUI), 12, 127
Grounded theory, 42, 147
Grundfos, 9

H
Hardware-in-the-loop (HIL), 211
Home automation, 143
HYPEX model, 12, 155–163
Hypothesis generation, 161

I
"If it hurts, do it often", 7
Innovations, 16, 35
Innovation system, 25–26
Innovativeness, 90
Integration flows, 110
Interaction challenges, 51
Interaction speed, 59–64
Internet of Things, 5
Inter-organizational relationships, 15
Inter-personal conflicts, 63
Inter-team documentation, 78–79

J
JAutomate, 134
Jeppesen, 9

L
Large-scale embedded systems, 83–92
Legacy functionality, 100

M
Malmö University, 9
Manual testing, 129
Measurement information model, 185
Mechatronics, 211–222
Meta-model, 110
Military defense, 20
Model based testing, 130, 211
Model driven engineering (MDE), 13, 214
Multi-disciplinary teams, 6
Multiple case study, 147

N
Navigable documentation, 79
Non-functional requirements (NFR), 170

O
Open innovation, 16
Operational data, 150
Operators, 204
Opportunity-based development, 90
Organization, 16
Organizational boundaries, 57–59, 80
Organizational performance metrics, 12

P
Paradigm shift, 16
Pattern distillation, 44
Performance profiles, 167
Periodicity of testing, 101
Platforms, 11, 213
Plenary sessions, 44
Politicized prioritization process, 160
Post-deployment data collection, 143–153
Practices, 15
Pre-release properties, 167
Processes, 15
Product
  management, 17, 85
  performance, 167
  sales, 85
  variation, 84
Project
  leaders, 20
  management, 74
PROPS, 205

Q
Quality assurance (QA), 20
Quality attributes, 100, 104
Questionnaires, 44

R
R&D accuracy, 155
R&D efficiency, 155
Refactoring, 49
Reference architecture, 74
Release cycle, 85
Repairing, 194–195
Replication speed, 60
Requirements engineering, 150
Research design, 41–42
Responsiveness, 49, 85
Retrospective sessions, 46, 107
Risk management, 43
Roadmap teams, 86
Root factor analysis, 60
Root factors, 60

S
Saab Electronic Defense Systems, 9
Scale, 85
Scope of testing, 100
SCRUM, 73
Scrum, 39, 206
Security, 143
Self-driving car, 117
Self-healing measurement systems, 183–199
Self-managed teams, 40
Semi-structured interviews, 20
Sikuli, 133
Simulation-based testing, 123
Simulink, 216
Social network systems, 145
Software architects, 20
Software-as-a-Service (SaaS), 3, 145
Software center, 9–10, 31
Software engineering, 13
Software-intensive organizations, 143
Software-intensive systems industry, 4–6
Software platform, 11
Software reuse, 11
Speed, 11, 16
Sprints, 10, 17, 46
Stairway to Heaven, 7, 15–26
Stakeholders, 16
Steering Committee, 31
Steering group, 31
Streamline development, 204
Surveillance, 20
System engineers, 20
System management (SM), 170
Systems engineering (SE), 20
System under test (SUT), 127

T
Task force group, 31
Team architect, 39
Team interactions, 51–65
Technical debt, 49
Technology, 16
Telecommunication, 143

Telecommunication systems, 21
Telecom networks, 204
Testability, 45
Test automation, 128
Test case generation, 130
Test-driven development (TDD), 7, 99
Test strategy, 98
Threat detection, 20
Transportation, 143
Transport solutions, 20

U
Unit test suite, 109
University of Gothenburg, 9

V
Validation, 6
Verification, 6
Version control, 84
Virtual test environments, 118
Virtual testing, 122–125
Virtual verification, 217
Visual GUI testing (VGT), 127–139
Volvo, 9
Volvo Car Corporation, 9, 180

W
Waterfall, 17
Waterfall process, 219
Ways of working, 203–210
Web 2.0 technologies, 3, 145
Widget based GUI based testing, 131

X
"10X in 10 years", 9