Advances in Using Agile and Lean Processes for Software Development

Pilar Rodríguez, Mika Mäntylä, Markku Oivo, Lucy Lwakatare, Pertti Seppänen, Pasi Kuvaja

Abstract

Software development processes have evolved according to market needs. The fast changing and unpredictable conditions that characterize current software markets have favoured methods advocating speed, flexibility and efficiency in software industry. Agile and Lean software development are in the forefront of these methods. Likewise, as the interest in Agile and Lean increases in software industry, so does the attention from the research community on these methods. This chapter presents a unified view of Agile software development, Lean software development and the most recent advances towards rapid releases, continuous delivery and continuous deployment. First, we introduce the area and explain the reasons why the software development industry begun to move into this direction in the late 90's. Then, in Section 2, we characterize the research trends on Agile software development over the years. This section helps the reader understand the relevance of Agile software development in the research literature. In Section 3, we provide a walk through the roots of Agile and Lean thinking, as they originally emerged in manufacturing. This walkthrough explains the theoretical differences on the basis of how these paradigms have evolved and have been interpreted in the software domain. Section 4 provides a detailed understanding on the way in which Agile and Lean processes are used for software development. In particular, the main characteristics, and most popular methods and practices of Agile and Lean software development are developed in this section. Section 5 is fully dedicated to rapid releases, continuous delivery and continuous deployment, the latest advances in the area to get speed. The concept of DevOps, as a means to take full (end-to-end) advantage of Agile and Lean, and Lean start-up, as an approach to foster innovation in software products, are the focus of attention of the two following sections, Section 6 and Section 7 respectively. Finally, the last section of the chapter is dedicated to two aspects of Agile and Lean software development that are important in practice: 1) metrics that are used to guide decisions in the context of Agile and Lean; and 2) technical debt as a mechanism that, when properly applied, can be used as a means to gain business advantage in an Agile context. To wrap up the chapter, we peer into future directions for Agile and Lean software development processes.

Keywords: software processes, agile software development, lean software development, lean thinking, le-agility, rapid releases, continuous delivery, continuous deployment, DevOps, lean startup, metrics, technical debt.

1. Introduction

This chapter will provide you with advanced knowledge on Agile, Lean and the latest progress towards Rapid software development. You may wonder, *why should I care about software processes, and, in particular, about Agile and Lean software development?* Indeed, one may legitimately question the importance of software development processes as many successful software innovations have not been *initially* held by well-defined software processes. Take, for example, the case of Facebook. Facebook, which emerged as a student's project at Harvard University in 2004, is currently one of the world's most popular social network (Wasserman, 2013). However, as companies grow, business and technical risks also increase. More customers become dependent on the developed products, projects become larger, the development process involves more people and aspects such as work coordination become essential (Münch et al., 2012). Using the example of Facebook, what was initially composed of a group of students, rapidly became one of the Fortune 500 companies, with a revenue of US$27.638 billion in 2016. Today, Facebook has around 18770 employees and over 1.94 billion monthly active users. Obviously, nowadays, Facebook needs to coordinate its work through a more sophisticated development process, which is, indeed, based on Agile and Lean software development[1].

You could also wonder, *then, from the wide umbrella of software development processes that are available in the literature, which is the best one I should apply? Why should I care about Agile and Lean?* The answer to this question is not straightforward, and it will depend on your company's business context and organizational needs. The traditional phase-gate process, commonly applied in other engineering fields, has been widely used in software development for decades (e.g. Waterfall model; Royce, 1970). However, nowadays, software has acquired an essential role in our lives, which has completely changed the way in which the software development business is run these days. In a software driven world, innovation, speed and ability to adapt to business changes have become critical in most software business domains. If this is your case, you can greatly benefit from Agile and Lean software development processes, like companies such as Facebook, Microsoft, IBM, Google, Adobe, Spotify, Netflix... just to mention a few, already did. Next, we briefly describe how software processes have evolved over time to better understand the relevance of Agile methods and, also, the reasons why they caused some initial controversy in the software community.

1.1. Evolution of software development processes

The Software Engineering literature has stressed the importance of software development processes and their influence on product quality for decades (Oivo et al., 1999; Münch et al., 2012). Consequently, efforts for improving software development processes have made different contributions over time. Among the landscape of contributions to software processes, we can find process models such as the Waterfall model (Royce, 1970) or the Spiral model (Boehm, 1988), process standards, which are widely used in the industry, such as the ISO/IEC 12207:2008 (2008) and ISO/IEC 90003:2004(E) (2004), model-based improvement frameworks such as CMMI (2010), SPICE (ISO/IEC 15504:1998) and Bootstrap (Kuvaja and Bicego, 1994), and, more recently, adaptive approaches for software development such as Agile and Lean methods (Beck et al., 2001). Figure 1 depicts a timeline with some of these important contributions to software development processes[2]. It bases on the review of process models conducted by Münch et al. (2012).

[1] *Scaling Facebook to 500 Million Users and Beyond,* Facebook engineering web page, http://www.facebook.com/note.php?note_id=409881258919 (last accessed, 15/06/2017).
[2] The purpose of Figure 1 is to show trends in software development processes. Therefore, it includes representative contributions with illustrative purposes. Classical software engineering books such as (Sommerville 2010) and guides such as the *Software Engineering Body of Knowledge* (SWEBOK) provide a more complete inventory on software engineering processes and related topics.

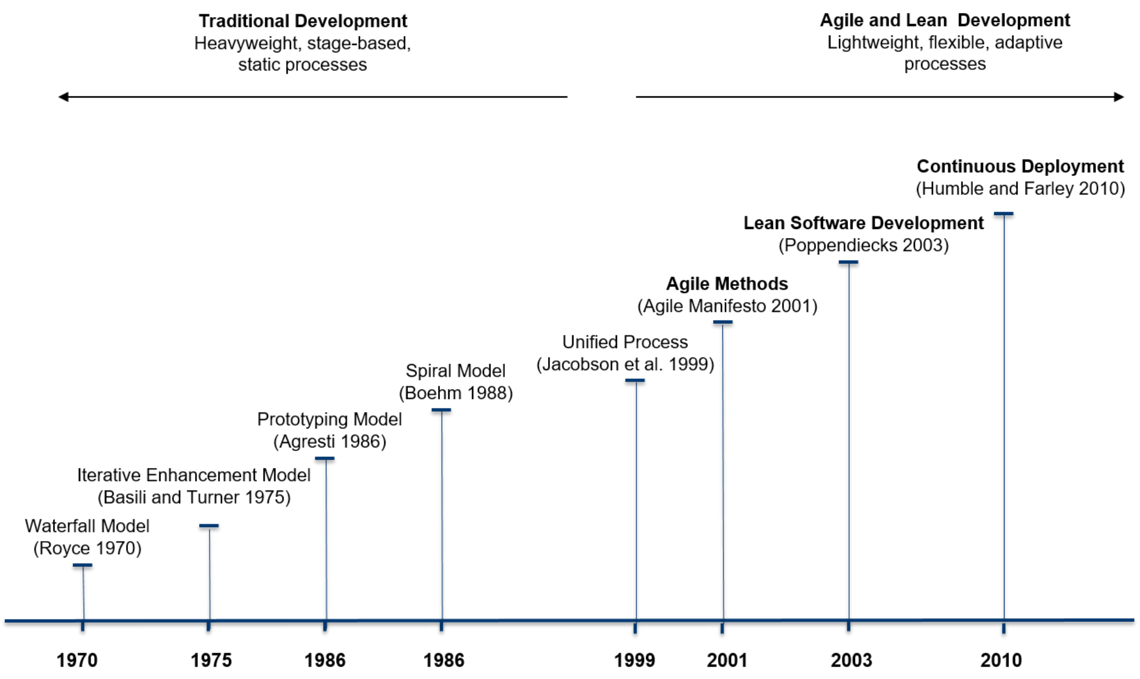


Figure 1. Contributions to software development processes over the last four decades.

Software development processes have gradually evolved from heavyweight, stage-based processes, which split the development process on distinct sequential phases and built on the assumption that requirements are relatively stable, (such as the Waterfall model (Royce, 1970)), to lighter/more flexible processes that try to cope with current business turbulence. As illustrated in Figure 1, process-centric models, which were considered deterministic and repeatable (Osterweil, 1987), over time have given way to flexible approaches, which emphasize speed, continuous evolution of software functionality, flexibility, face-to-face communication, customer involvement and a minimal number of lightweight artifacts.

1.2 The emergence of Agile software development

The environment in which the software development industry works today, which is fundamentally different from the environment in which it worked twenty years ago, is a main driver behind the evolution of software processes towards Agile and Lean software development. On the one hand, the prevalence of software products has shaped a software driven society. Software is all around us, from the (electronic) toothbrush that we use in the morning, to the alarm that we set-up before going to sleep. Thus, the business around software products is open to countless opportunities. On the other hand, the *Age of Information*[3], which is strongly based on the Internet, has made digital resources constantly available for everyone. Information flows are accelerated, global communication and networking are faster and individuals can explore their personal needs easier. Overall, software relevance, globalisation and market dynamics, characterised by rapid and unpredictable change, have shifted the software development landscape. These market features put pressure on software and software-intensive organisations to develop what Eisenhardt and Martin (2000) called '*dynamic capabilities*'. Software development organisations need to be creative, innovative and flexible, while working with incomplete information and pursuing economic efficiency. This situation is particularly relevant for organisations in domains such as web or mobile applications and services. In these contexts, applications are frequently developed in a matter of weeks or months, rather than years, as it was common when using traditional processes (Wasserman, 2013). Yet, the emergence of trends like internet of things (IoT) will shape software development processes of more traditional industries in the near future.

To cope with this situation, a community of practitioners formulated the Agile Manifesto in 2001 (Beck et al., 2001). In a two days meeting in the Wasatch mountains of Utah in February 2001[4], fourteen independent software practitioners proposed the Agile Manifesto as an alternative to software processes that they considered too inflexible and slow to adapt to business changes. It provoked a change in the way in which the Software Engineering community was addressing

[3] *Information Age* refers to the age that began in the 1970s in which society started to work in networks through a constant flow of information based on technology. Since then, company competitiveness is very dependent on its knowledge on technology, information and network access (Castells 2011).

[4] http://agilemanifesto.org/history.html

software development processes. In essence, Agile recognises the limitations of anticipating external business changes and provides means oriented to implement minimum viable products quickly as a key competitive advantage (Abrahamsson et al., 2002). The idea is that minimum viable products are incrementally developed in collaboration with customers, as more information about the product is known. Contrary to traditional phase-based software development processes, Agile appraises iterative development. Thus, traditional software development phases (e.g. requirements, development, testing, etc.) are run in small cycles to achieve continuous delivery and feedback and, therefore, improve the company's capability to adapt to market and customer fluctuations (Abrahamsson et al., 2002). Further, Agile emphasizes the human side of the socio-technical software development activity as a primary driver of project success (Conboy et al., 2011; Cockburn and Highsmith, 2001). Agile methods advocate that the key role that people plays in software development makes the development process less deterministic and repeatable than previously considered by traditional prescriptive methods (Osterweil, 1987). Kruchten (2011) explained it using the party metaphor: '*Consider you organize a party this Saturday at your home. You invite a group of people, order the food, move the furniture, and you record carefully: who comes, when, who eats what, drinks what, etc., and everyone leaves having had a good time. The next Saturday, to have another great time, you decide to have the very same people, the very same food and drinks, you ask them to dress the same way, arrive at the same time, you introduce them to the same people, bring them the same glasses with the same drinks, make the same jokes,... will you be guaranteed to have a nice party again...?*'. In the same line, Boehm stressed that '*one of the most significant contributions of the agile methods community has been to put to rest the mistaken belief that there could be a one-size-fits-all software process by which all software systems could be developed*' (Münch et al., 2012). The bad news is that, unfortunately, there is no silver bullet for software development processes. That is, there is no definitive answer to the question: *what is the best software development process I should apply?* However, the good news is that Agile and Lean software development offer means to cope with current software business dynamics, which can be adapted to different context.

Regarding to the adoption level of Agile and Lean software processes, what was initially considered as a fad has been progressively consolidated. Nowadays, a great extent of the software industry follows these methods. For example, the latest State of Agile Survey by VersionOne (2016) found that 43% of its 3,880 respondents worked in software organizations with more than 50% of their teams using Agile. Only 4% of respondents indicated that in their organization no team was using Agile. Similarly, a scientific study conducted in Finland in 2012 concluded that the Finnish software industry has moved towards these methods, being the 58% of the survey's respondents working in organizational units using Agile software development (Rodríguez et al., 2012). Nevertheless, it is also worth mentioning that the adoption of Agile has not always been a smooth process. Indeed, Agile software development caused some initial controversies among those that saw it as an attempt to undermine the software engineering discipline (Rakitin, 2001). Particularly, there were some initial concerns about using it as an easy excuse to avoid good engineering practices and just code up whatever comes next (Rakitin, 2001; Boehm, 2002; Beck and Boehm, 2003; Rosenberg and Stephens, 2008). However, as we will see in the rest of the chapter, and particularly in sections 4 and 5, Agile software development is a much more disciplined approach than it may seem at first sight.

1.3 Lean thinking and continuous deployment

A few years after the Agile Manifesto, Lean thinking started to become also the center of attention of the software development industry (Poppendieck and Poppendieck, 2003). However, Lean did not appear as a software development approach of its own until 2003, when Mary and Tom Poppendieck introduced it in their book *Lean Software Development: An Agile Toolkit* (Poppendieck and Poppendieck, 2003). One of the reasons that captured the attention of the software community upon Lean thinking was the focus of Lean on the organisation as a whole (Poppendieck and Poppendieck, 2003; Larman and Vodde, 2009; Vilkki, 2010; Laanti, 2012). Agile software development was mainly conceived to work for small software development teams (Dybå and Dingsøyr, 2008). However, scaling Agile methods, such as Scrum (Schwaber and Beedle, 2002) or eXtreme Programming (Beck and Andres, 2004), to operate a whole organisation was not straightforward, since Agile does not explicitly provide support in that sense (Turk, France and Rumpe, 2002; Abrahamsson et al., 2009; Maples, 2009; Vilkki, 2010). Thus, Lean thinking was discerned as a promising solution to be combined with Agile methods. Nowadays, Lean thinking is not only applied as a way of scaling Agile, but as well as a way of improving software development processes from a wider perspective (Maglyas et al., 2012).

The interest on Lean thinking, which originally emerged in the automotive industry (Womack, Jones and Roos, 1990), is not exclusive to the software development industry. Lean thinking has been widely adopted in different domains, from the healthcare domain to the clothing industry. Table 1 shows some examples of benefits that are attributed to Lean in diverse domains, in terms of profitability, productivity, time-to-market and product quality.

Table 1. Examples of benefits attributed to Lean thinking in companies from different domains.

Aspect	Benefits
Profitability, productivity	Business Week reported that Toyota[5] cut \$2.6 billion out of its \$113 billion manufacturing costs, without closing a single plant in 2002 (Bremner et al., 2003). Although Toyota has suffered challenging crisis in its history (Cusumano, 2011), the company led global automobile sales until the March 2011 earthquake in Japan, which brought the production to a halt (OICA).
Time to market	Zara's business model, operating in the retail clothing industry, is based also on Lean thinking. According to Tokatli (2008), Zara significantly reduced its lead-time by collecting and sharing input from customers daily and using Lean inventories. Shorter lead times enabled Zara to deliver new items to stores twice a week (as much as 12 times faster than its competitors) and to bring in almost 30000 designs each year, as opposed to the 2000–4000 new items introduced by its competitors (Tokatli, 2008). In the healthcare domain, the application of Lean has reported significant improvements in reducing patient waiting lists, floor space utilisation and lead-time in laboratorial tests (de Souza, 2009).
Product quality	Toyota is also well-known as an icon in quality control (Cusumano, 2011); for example, the Toyota Lexus CT200h received the maximum rating under the Japanese overall safety assessment in 2011 (2011 Japan New Car Assessment Program).

Nowadays, there is a vogue for rapid and continuous software engineering. It refers to the organisational capability to develop, release and learn from software in rapid parallel cycles, often as short as hours or days (Mäntylä et al,. 2015; Fitzgerald et al., 2017; Rodríguez et al., 2017). *Speed* is the key aspect in this trend. The aim is to minimize the step between development and deployment so that new code is deployed to production environment for customers to use as soon as it is ready. Continuous deployment (CD) is the term used to refer to this phenomenon (Humble and Farley, 2010; Olsson et al., 2012 ; Fitzgerald and Stol, 2014; Järvinen et al., 2014; Claps et al., 2015). Although the concept of deploying software to customers as soon as new code is developed is not new and it is based on Agile software development principles, CD extends Agile software development by moving from cyclic to continuous value delivery. This evolution requires not only Agile processes at the team level but also integration of the complete R&D organization, parallelization and automation of processes that are sequential in the value chain and constant customer feedback. Rapid releases, continuous delivery and continuous deployment will be widely discussed in Section 5.

2. Trends on Agile, Lean and Rapid Software Development

Given the importance of Agile, this section provides a more detailed account of the history and topics of Agile in the scientific literature. The analysis helps the reader understand how a movement initially promoted by practitioner's proponents of Agile methodologies has become a well-established research discipline. Particularly, we analyze the various concepts that are investigated inside the scientific literature of Agile through the lens of automated text mining of scientific studies of Agile development. To characterize the research on Agile software development, we collected and analyzed a large set of studies by searching abstracts in the Scopus database, starting from the year 2000. Using search words "agile" and "software" resulted in over 7,000 published articles that we used in our trend mining.

2.1 Overview of Agile literature

First, we explore the research topics in Agile over the past two decades. We divided the articles into four roughly equally large groups based on the year of publication and plotted a word comparison cloud of their titles. Figure 2 shows the word comparison cloud. Early years (2000-2007) are in black, 2nd period (2008-2011) is in grey, 3rd period (2012-2014) is in red and the most recent era (2015-2017) is in blue. The words in each quadrat are more common in their given periods than in other periods, i.e. the size of each word is determined by its deviation from the overall occurrence average. The figure was plotted with R Package 'wordcloud'.

[5] Toyota and its Toyota Production Systems are well known as icons of Lean thinking.

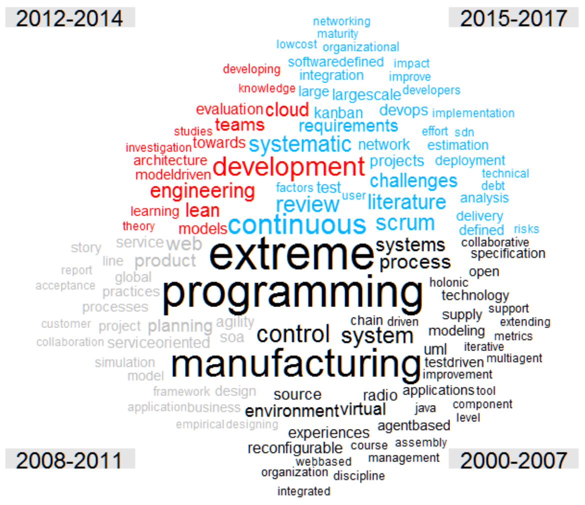


Figure 2. Word comparison cloud of research on Agile software development from 2000 to today.

Although the early years contain several articles about Agile software development in general, e.g (Highsmith and Cockburn, 2001; Boehm, 2002; Reifer, 2002), the Agile software development method that was the most investigated early was Extreme Programming (Beck and Andres, 2004). Extreme Programming is an Agile method with the most doctrinaire approach by setting detailed development practices such as test-driven development and pair-programming. The early success of Extreme Programming as well as declining interest towards it later, may well be explained by its mandating approach to a set of development practices. Adopting it at first can be straightforward as it gives clear rules such that all the development code is done by pair-programming. However, later people may realize that such ultimatums might not make sense in their context and, thus, they would turn to more flexible Agile methods. For example, for pair-programming research shows that having two junior programmers can form a cost effective pair while the same does not hold for two expert programmers (Arisholm et al., 2007). During the early years, many papers about Agile manufacturing appeared (Chen, 2001) which addressed the manufacturing of physical products. Work in this area also influenced Agile software development as we shall explain in the next section.

In the 2nd period (2008-2011) research about Agile software development in the technological contexts of the web and service-oriented architectures (SOA) increased in numbers. Thus, the scientists working on technical advances also connected their works to software process advances in Agile software development, e.g. (Perepletchikov et al., 2010). During that time, researchers connected Agile software development to work on Global and Distributed software development, which addresses issues of developing the same software in multiple sites with diverse locations around the globe (Hossain et al., 2009). You should also notice that papers connecting Agile to Global and Distributed software development were the first research steps towards scaling Agile beyond one co-located team. Research efforts in Agile software development increased also in the areas of process simulations (Port and Bui, 2009), linking business to Agile (Vähäniitty and Rautiainen, 2008), and in customer and other type of collaborations (Hoda et al., 2011).

In the 3rd period (2012-2014) the technical advances of Cloud computing (Delen and Demirkan, 2013) and Model-driven development (Rivero et al. 2014) were linked with the process advances of Agile. The term “Lean” gained popularity as a way to advance Agile even further (Wang et al., 2012) Learning and teaching Agile was studied in universities (Scott et al., 2014) but articles about learning Agile also include papers about self-reflection in Agile teams (Babb et al., 2014).

In the most recent period (2015-2017), we can see the rise of word Continuous, which increases in popularity due to the practices of continuous integration, continuous deployment, and continuous delivery (Leppänen et al., 2015; Rodríguez et al., 2017). These practices highlight the importance of running working software, which is integrated and delivered to clients continuously, rather than providing major updates biannually for example. We also notice increase in popularity of Kanban that is a method for managing work and tasks in Agile software development. Systematic Literature Reviews about Agile increased in popularity in the most recent period as well. SLRs offer an overview of particular areas of Agile software development like using software metrics in Agile (Kupiainen et al., 2015) or technical debt (Behutiye et al., 2017).

2.2 Trends in Agile

The previous section offered a general overview on Agile software development over the past two decades. For this section, we did individual searches on topics presented in previous section but also on engineering topics we knew that had been introduced with Agile development such as refactoring. Thus, we searched for the keywords shown in Figure 3 combined with the word "software". Adding word software was necessary to exclude, for example, papers discussing Scrum in context of Rugby instead of Agile software development. Research in software engineering in general has increased in past two decades (Garousi and Mäntylä, 2016) so we normalized all search results by the number of papers found with the general terms of "software development" or "software engineering". We also normalized all trend graphs to have values between 0 and 1 where max is the year with the highest normalized amount of papers.

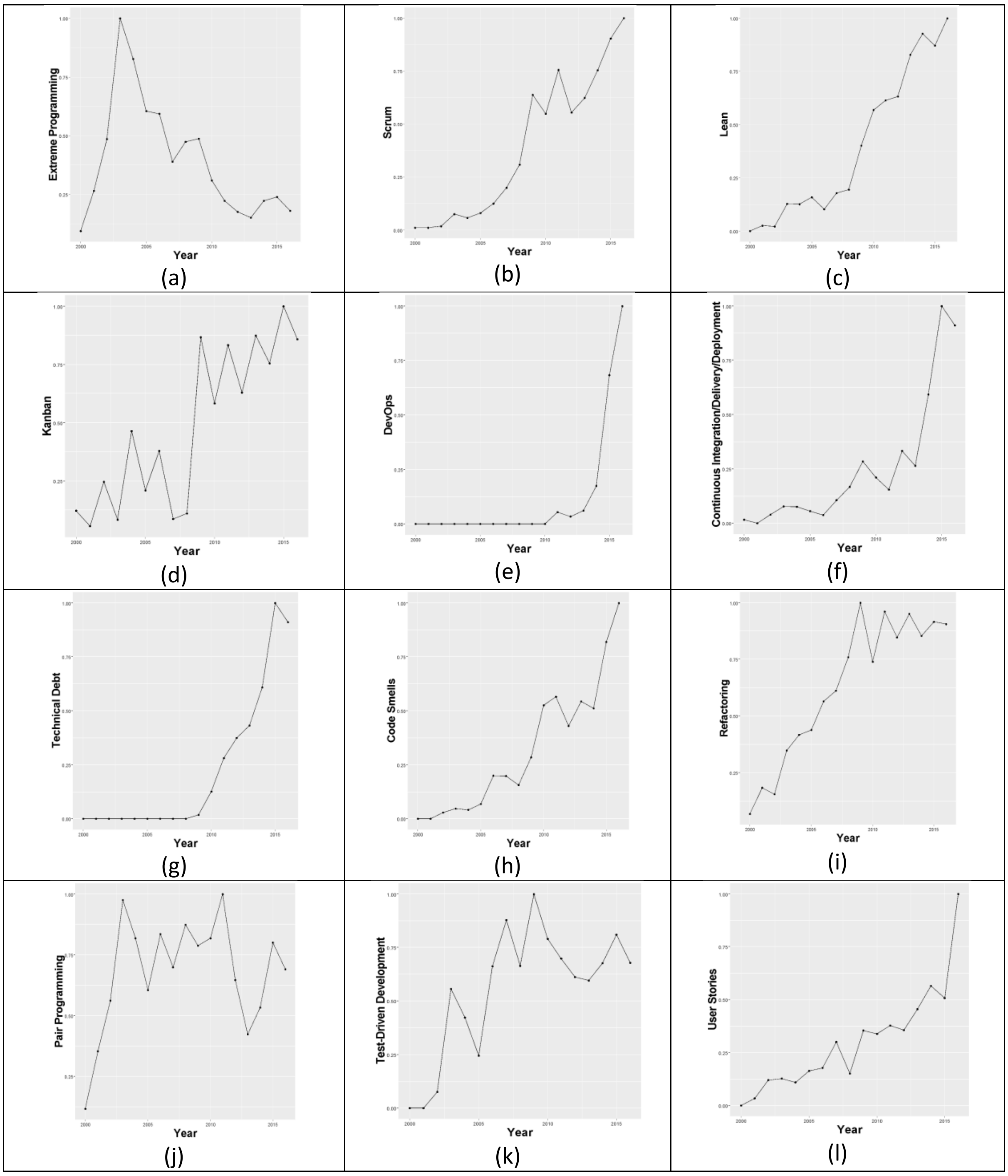


Figure 3. Trends on Agile Software Development.

The concepts of Extreme Programming, Scrum, Lean, Kanban, DevOps, and Continuous Integration/Deployment/Delivery are shown in Figure 3 (a)-(f). We can see that XP has been declining in research interest since 2004, Figure 3 (a). Scrum, Lean and Kanban have had good popularity since 2009 but they still appear to be gaining popularity, although the growth trends is not as steep as it has been in the past. DevOps and Continuous Integration/Deployment/Delivery seem to be gaining popularity fast and can be characterized as the hot research topics in Agile software development.

Figure 3 (g)-(i) show the popularity of the concepts of Technical Debt and Code Smells that are used to describe suboptimal software engineering solutions which hinder the velocity of Agile teams. We can see that research in both areas is still increasing. The concept of Technical Debt appears to be more recent than Code smells. The concept of Technical Debt is also larger than code smells. Code smells refer to code problems only, while technical debt can refer to problems with using an old version of the build server for example. Refactoring is about code modifications to fix Code Smells or Technical Debt or due to the need for other behavior persevering changes. The popularity of refactoring research has been stable since 2009.

Finally, Figure 3 (j)-(l) show the popularity of three Agile software development practices that were popularized by Extreme programming that are Pair-Programming, Test-Driven Development and User-Stories. We can see that although the popularity of Extreme Programming has declined in research literature, the popularity of Pair-Programming has remained relatively stable since 2003 and research interest in TDD has similarly remained stable since 2007. Research on User Stories that could also be titled as Agile requirements engineering is still experiencing fast growth. So, even when Extreme Programming might become extinct, some of its practices are still sparking wide interest.

2.3 Summary

This section has demonstrated popularity of many Agile software development concepts and practices in research literature. Next sections will go deeper to the foundations of Agile and help the reader in putting these concepts together as all of these concepts are essentially solutions of the business and engineering problems companies are having while developing software based products, services and solutions.

3. A Walk through the Roots of Agile and Lean Thinking

This section describes Lean thinking, agility and its combination (known as Le-agility) as they originally arose in manufacturing (more concretely, in the automotive industry). Understanding the principles of Agile and Lean in the manufacturing domain helps better understand the essence of these paradigms. This section provides background of Agile and Lean that we believe may be interesting for the reader, although, its reading is not mandatory to understand the rest of the chapter.

3.1 Lean Thinking

Lean was born as part of the industrial renaissance in Japan after the *Second World War* in the 1940s. Traditional mass production - which is based on producing large batches of products through continuous assembly lines of interchangeable parts - was unsuitable in a Japanese market lashed by the war and characterised by low volumes. Thus, Japanese manufacturers found themselves surrounded by obstacles that forced them to develop new methods. Japanese new production ideas led to what we know today as Lean or Lean thinking. However, the term Lean was not used until researchers from the Massachusetts Institute of Technology (MIT) began research under the International Motor Vehicle Program (IMVP) in 1986[6]. With the aim of investigating the differences across worldwide automotive industries, IMVP researchers discovered that the Japanese auto industry was far ahead when compared with the automotive industry in America. By carefully studying Japanese methods, particularly those of Toyota under its Toyota Production System (TPS), they conceived an entirely different production system, which they called Lean manufacturing (Womack et al., 1990). Thus, Lean or Lean thinking is the English term that western researchers used to describe Japanese original production methods.

According to MIT' researchers**, Lean is about '*doing more with less*' by ideally producing '*the right things, at the right time and in the right place*'** (Womack et al., 1990). But, how to do more with less? Based on fundamental industrial engineering principles, Lean thinking, as interpreted by MIT's researchers, steeps in a philosophy of maximizing value and minimizing waste. **Five principles guide this philosophy** (Womack and Jones, 1996):

[6] *International Motor Vehicle Program* (IMVP, http://www.imvpnet.org/) is an international network that focuses on analyzing the challenges faced by the global automotive industry. IMVP, founded at the Massachusetts Institute of Technology (MIT) in 1979, has mapped Lean methodologies, established benchmarking standards and proved the entire automotive value chain.

- ***Value*** is defined as everything that a customer is willing to pay for. Defining and understanding value from the perspective of the customer is the central focus of Lean thinking. Lean thinking stresses that every activity inside a company should contribute to create customer value. The goal is to make organisations deliver as much customer value as possible. Its counterpart, waste, or *muda* in Japanese, is everything that consumes resources but creates no customer value.
- ***Value stream*** is an optimised end-to-end collection of actions required to bring a product/service from customer order to customer care. Lean thinking considers three types of activities in a stream: 1. activities that create value and, therefore, are legitimately part of the value stream; 2. activities that create no value for the customer but are currently needed to manufacture the product. These activities remain in the value stream as, otherwise, the product cannot be manufactured, but the aim is to improve the production process so that they can be removed at some point; and 3. activities that clearly create no value for the customer and should be immediately removed from the value stream.
- ***Flow*** means that production activities are organised as a continuous 'flow', eliminating discontinuities. Flow requires that unnecessary steps and inventories are eliminated (waste). Opposite to mass production, in which interchangeable parts are continuously produced to assemble final products, in Lean thinking, products are made using 'single piece flows'. In a single piece flow, operations are efficient in a way that batch sizes and inventories are replaced by working on one product at a time.
- ***Pull*** implies producing products only when they are really needed. Accordingly, in a pull system, an upstream process only produces when a downstream process is ready and 'pulls' some more work from the upstream process. This minimizes inventories. In traditional push systems, work is produced in response to a pre-set schedule, regardless of whether the next process needs it at that time. Accordingly, the process is transformed from batch-and-queue to just-in-time (JIT) production.
- ***Perfection***, or Kaizen in Japanese, refers to the enterprise-level improvement and learning cycle. Lean aims to achieve zero waste and defects based on the concept that there is no end in the strive for perfection.

MIT's five principles of Lean are probably the most popular interpretation of Lean thinking. Some voices have questioned the validity of IMPV's studies by pointing out to a certain biased interpretation of the original Japanese methods (Dybå and Sharp, 2012). For example, Toyota and its TPS, which had a great influence on Lean thinking, was originally called the *Respect for Humanity System*. The Respect for Humanity System was based on two main pillars: *continuous improvement* and *respect for people* (Fujimoto, 1999). Thus, value and waste were not originally the focuses of the paradigm. When later Lean was interpreted from a western point of view, influenced by the existing western's social-economical system, value and waste became key components of Lean thinking. Nevertheless, biased or not, it is indisputable that MIT's studies, and the five principles outcome of these studies, have vastly influenced the shape of Lean thinking as it is understood nowadays. The book *'The machine that changed the world'*[7] (Womack et al., 1990) provides a more detailed description of the story of Lean as interpreted by MIT's researchers.

Besides MIT's researchers, other authors have also provided slightly different interpretations of Lean thinking (see Table 2). Thus, Lean has incrementally evolved (Ohno, 1988; Womack et al., 1990; Womack and Jones, 1996; Liker, 2004; Morgan and Liker, 2006). As presented in Table 2, Onho (1988) described the Toyota Production System (TPS) mainly from a perspective of reducing waste (e.g. overproduction, waiting, transportation, over-processing, inventory, movement and defects). Although Onho emphasized also the concepts of Just-in-Time and autonomation, many authors have taken waste as the key element of Lean thinking. For example, Cusumano referred to Lean as: '*The authors* [referring to the authors of The machine that changed the word] *used the term 'Lean' to describe any efficient management practice that minimizes waste*' (Poppendieck and Cusumano, 2012). In a similar vein, Shah and Ward defined Lean production as '*an integrated socio-technical system whose main objective is to eliminate waste by concurrently reducing or minimizing supplier, customer and internal variability*' (Shah and Ward, 2007). The excessive focus on waste has frequently created certain unfair reputation to Lean thinking among those who have seen it as an easy excuse to get rid of employees and resources. However, although the concept of waste is undoubtedly important in Lean, the five, fourteen and thirteen principles of Lean thinking by Womack and Jones (1996), Liker (2004) and Morgan and Liker (2006) respectively, clearly show that Lean thinking is more than only removing waste. As we will see in Section 4, the interpretation of Lean thinking in software development has been mainly taken from an Agile perspective (Lean software development was built upon the values and principles formulated in the Agile Manifesto). Thus, although waste is important in a software development context as well, Lean software development goes way beyond waste reduction.

[7] As an indicator "*The machine that changed the world*" (Womack *et al.*, 1990) is one of the most widely cited references in operations management (referenced by 15002 sources in Google Scholar, accessed 09.09.2013).

Table 2. Main sources of Lean thinking in a manufacturing context.

Source	**Description**
Ohno (1988)	Description of the Toyota Production System (TPS). ● The basis of the TPS is the absolute elimination of waste. Ohno describes seven classes of waste: *overproduction* (producing more quantity of product or product's components than needed), *waiting* (waiting for a previous upstream activity to be done), *transportation* (unnecessarily moving of materials), *over-processing* (unnecessarily processing by a downstream step in the manufacturing process), *inventory* (storing of materials that are not needed for the present), *movement* (unnecessary motion of people) and *defects* (rework because of quality defects). ● According to Ohno, two pillars support this system: 1) *Just-In-Time* (JIT), which refers to producing the right items at the right time in the right amounts, and 2) *autonomation*, which refers to a quality control principle that provides machines and operators the ability to stop the work immediately whenever a problem or defect is detected.
Liker (2004)	Liker considers fourteen principles in his interpretation of Lean thinking as follows: 1. Base your management decisions on a long-term philosophy, even at the expense of short-term financial goals. 2. Create a continuous process flow to bring problems to the surface. 3. Use 'pull' systems to avoid overproduction. 4. Level out the workload (heijunka). 5. Build a culture of stopping to fix problems, to get quality right the first time. 6. Standardized tasks and processes are the foundation for continuous improvement and employee empowerment. 7. Use visual control so no problems are hidden. 8. Use only reliable, thoroughly tested technology that serves your people and process. 9. Grow leaders who thoroughly understand the work, live the philosophy, and teach it to others. 10. Develop exceptional people and teams who follow your company's philosophy. 11. Respect your extended network of partners and suppliers by challenging them and helping them improve. 12. Go and see for yourself to thoroughly understand the situation (genchi genbutsu). 13. Make decisions slowly by consensus, thoroughly considering all options; implement decisions rapidly (nemawashi). 14. Become a learning organisation through relentless reflection (hansei) and continuous improvement (kaizen).
Morgan and Liker (2006)	Toyota Product Development Process - Three categories and thirteen principles: Process 1. Establish customer-defined value to separate value-added from waste. 2. Front-load the product development process to explore thoroughly alternative solutions while there is maximum design space. 3. Create a levelled product development process flow. 4. Utilize rigorous standardization to reduce variation, and create flexibility and predictable outcomes. Skilled people 5. Develop a chief engineer system to integrate development from start to finish. 6. Organize to balance functional expertise and cross-functional integration. 7. Develop towering technical competence in all engineers. 8. Fully integrate suppliers into the product development system. 9. Build in learning and continuous improvement. 10. Build a culture to support excellence and relentless improvement. Tools & Technology 11. Adapt technology to fit your people and process. 12. Align your organisation through simple, visual communication. 13. Use powerful tools for standardization and organisational learning.

3.2 Lean manufacturing toolkit

Lean thinking has no formal practices, but builds on any practice that supports its principles. Still, a toolkit of recommended practices, tools and techniques, from which to choose from, emerged to implement the fundamentals in practice. Table 3 summarises those that are relevant from a software development perspective, as they have been mentioned in the context of Lean software development (and will appear later on in the next sections). For a more thorough listing of Lean terms, with examples and illustrations, the reader is referred to The Lean Lexicon created by The Lean Enterprise Institute (Marchwinski and Shook, 2008).

Table 3. Selection of recommended practices/tools/techniques from Lean thinking.

Practice/Tool/Technique	**Description**
Autonomation (Jidoka)	Quality control principle that aims to provide machines and operators the ability to stop the work immediately whenever a problem or defect is detected. This leads to improvements in the processes by eliminating the root causes of defects. Related to the concept of 'Stop-the-line'.
Cell	Location of processing steps for a product immediately adjacent to each other so that parts can be processed in very nearly continuous flow.
Chief Engineer (Shusa)	Manager with total responsibility for the development of a product. The chief engineer is the person who has responsibility to integrate the development team's work around a coherent product vision.
Just-in-Time (JIT)	System for producing the right items at the right time in the right amounts.
Kaikaku	Process improvement through a radical change.
Kaizen	Continuous process improvement through small and frequent steps.
Kanban	Method based on signals for implementing the principle of pull by signalling upstream production.
Lead time	Metric defining the total time that a customer must wait to receive a product after placing the order.
Mistake-proofing (Poka-yoke)	Method that helps operators to prevent quality errors in production by, for example, choosing wrong parts. The method ensures that an operator can only choose the right action (e.g. product design with physical shapes that make impossible to wrongly install parts in wrong orientations).
Big room (Obeya)	Project leaders room containing visual charts to enhance effective and timely communications, and to shorten the Plan-Do-Check-Act cycle.
Set-Based Concurrent Engineering	Approach to designing products and services by considering sets of ideas rather than a single idea.
Self-reflection (Hansei)	Continuous improvement practice of looking back to think how a process can be improved.
Six Sigma	Management system focused on improving quality by using mathematical and statistical tools to minimise variability.
Standardization	Precise procedures for each activity, including working on a sequence of tasks, minimum inventory, cycle time (the time required to complete an operation) and takt time (available production time divided by customer demand).
Usable knowledge	Capturing knowledge to be applied in other projects.
Value Stream Mapping (VSM)	Tool for analysing value and waste in the production process. VSM provides a standardised language for identifying all specific activities that occur along the value stream.
Visual control	Tools to show the status of the system so that everyone involved in the production process can understand it at a glance.
Work-In-Progress (WIP)	Items of work between processing steps.
5-Whys	Method for analysing/finding the root cause of the problems consisting in asking 'why' five times whenever a problem is discovered.

3.3 Agility

Similar to Lean thinking, the agility movement has its origins in a manufacturing context as well. However, it emerged well after Lean thinking. Concretely, it appeared in manufacturing at the beginning of the 1990s, with the publication of a report by the Iacocca Institute (Nagel and Dove, 1991). This report defined **Agility as a solution to adjust and respond to change, and satisfy fluctuant demand through flexible production**. Specifically, Nagel and Dove defined Agility as '*a manufacturing system with extraordinary capability to meet the rapidly changing needs of the marketplace, a system that can shift quickly among product modes or between product lines, ideally in real-time response to customer demand*'. Christopher and Towill (2000) describe how '*the origins of agility as a business concept lie in flexible manufacturing systems, characterising it as a business-wide capability that embraces organisational structures, information systems, logistics, processes and in particular mind-set* […]'.

Agility seems to be '*highly polymorphous and not amenable to simple definition*' (Conboy, 2009). However, two aspects are usually stressed in the literature: *adaptation* and *flexibility*. The concept of adaptation is based on contingency theories, which are classes of the behavioral theory (Donaldson, 2001). These theories state that there is no universal way of managing an organization. The organizing style should be adapted to the situational environmental constraints in which the organization operates. Thus, organizations are seem as open systems that must interact with their environment to be successful. Successful organizations do not keep themselves in isolation but develop their adaptation capabilities. A close related term is organizational flexibility. Organizational flexibility is considered as '*the organization's ability to adjust its internal structures and processes in response to changes in the environment*' (Sherehiy et al., 2007). The aim of Agility is to achieve an adaptive and flexible organization able to respond to changes and exploit them to take advantage.

The literature highlights four capabilities that are particularly important to achieve an Agile organization (Sharifi and Zhang, 1999):

- ***Responsiveness*** is the ability to identify and respond fast to changes. It does not only mean identifying changes when they happen (being reactive to changes) but also proactively sensing, perceiving and anticipating changes to get a competitive advantage.
- ***Competency*** involves all capabilities that are needed to achieve productivity, efficiency and effectiveness. Competency comprehends each area in the organization, from strategic vision, appropriate technology and knowledge, product quality, cost effectiveness, operations efficiency, internal and external cooperation, etc.
- ***Flexibility*** refers to the organizational capability to make changes in products and achieve different objectives using the same facilities. Flexibility involves from product characteristics and volumes to organizational and people flexibility.
- ***Speed*** or quickness relates to making tasks and activities in the shortest possible time in order to reduce time-to-market.

To achieve these capabilities, Agile organizations are flat organizations with few levels of hierarchy, and informal and changing authority. Regulations with respect to job description, work schedules, and overall organizational policies are also loose, and trust is given to teams to perform their tasks. Teams are self-organized and job descriptions are redefined based on need (Sherehiy et al., 2007). A more comprehensible description of the main attributes of Agility as it originally emerged in the manufacturing domain can be found in (Sherehiy et al., 2007).

3.4 Combining Lean and Agile in Manufacturing

As in software development, the combination of Lean and Agile also occurred in manufacturing (Naylor et al., 1999). However, in a manufacturing context, the migration occurred from Lean and functional systems to Agile and customised production (Christopher and Towill, 2000). That is, Lean thinking was already in place in many organizations when they started to look for agility. Accordingly, the shift was characterised by a strong influence of Lean thinking, mainly understood as efficiency and waste reduction. In this context, Lean and Agile seemed to have some intertwined ideas (Naylor et al., 1999). Agile's focus on flexibility and capacity to embrace change could challenge Lean's focus on overall economic contribution. For example, achieving flexibility in a manufacturing context may request important investments in machines and storages. Marson-Jones et al. (2000)'s matrix guides the combination of Lean and Agile in a manufacturing context. The combination is guided according to the primary company's aim (see Figure 4). If the company mainly focuses on reducing cost (i.e. cost is the market winner and, therefore, dominates the paradigm as primer requirement), then Lean should be the primary strategy. In this case, quality, lead-time and service level are highly desirable but are not strictly requested. However, if the goal of the company is to improve its service level, Agile should take priority.

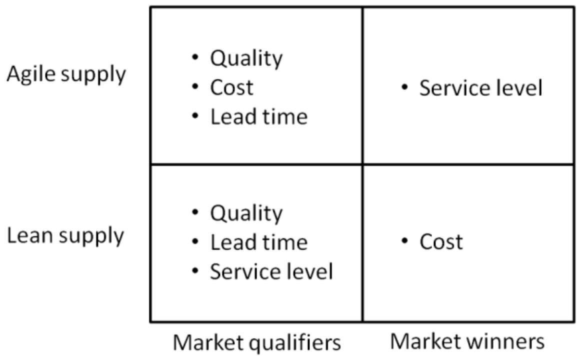


Figure 4. Marson-Jones et al.'s matrix to guide the combination of Agile and Lean in the manufacturing domain (source Marson-Jones et al. (2000)).

Moreover, the theory of Le-agility in a manufacturing domain (Naylor et al., 1999; van Hoek, 2000) says that whilst the combination of Agile and Lean might work well, since Lean capabilities can contribute to Agile performance, the combination has to be carefully planned to prevent risks that Agile's demands may cause on Lean capabilities (Naylor et al., 1999). Specifically, time and space restrictions are considered in the manufacturing domain, limiting the combination of Agile and Lean to three scenarios:

1. Different value streams, that is, different products.
2. The same product but at different points in time.
3. Using both paradigms at different points in the value stream by using de-coupling strategies (strategies to de-couple Agile and Lean).

In other words, in manufacturing, there is a tendency to suggest that although Agile and Lean can be combined, they cannot be used at the same time in the same space. Fortunately, the software domain does not have the same restrictions. Indeed, although building upon similar principles, the own characteristics of software development have made the combination of Agile and Lean in the software domain quite different from its combination in manufacturing. Next section develops this combination in detail.

4. Agile and Lean in Software Development

Section 3 has described the way in which Agile and Lean emerged in manufacturing. When it comes to the software development domain, we need to consider the fundamental differences between domains (Staats et al., 2011). For example, differently from manufactured products, software products are of an intangible nature, which makes it impossible to specify physical measures, affecting the entire concept of quality (Mandić et al., 2010). The concepts of value and waste are impacted as well. Software products are malleable and its value is not limited to a single time-bound effort. Defining value is not a straightforward tasks in software development (Mendes et al., 2017), having a whole research field on its own, value-based software engineering (Biffl et al., 2005). Similarly, waste does not have to follow necessarily the path of the original seven forms of waste as identified by Taiichi Ohno (Ohno, 1988). Most waste in manufacturing can be directly detected by observing the physical material flow and machine/worker activities. However, many times, intangible work items in software development challenge waste recognition and process improvement. Principles such as those related to flow or visual communication are also impacted due to this intangible nature. Another key difference is on the role of people. Whilst human presence in a manufacturing environment is mainly required to operate automated machines, in software development, people creativity and knowledge are essential.

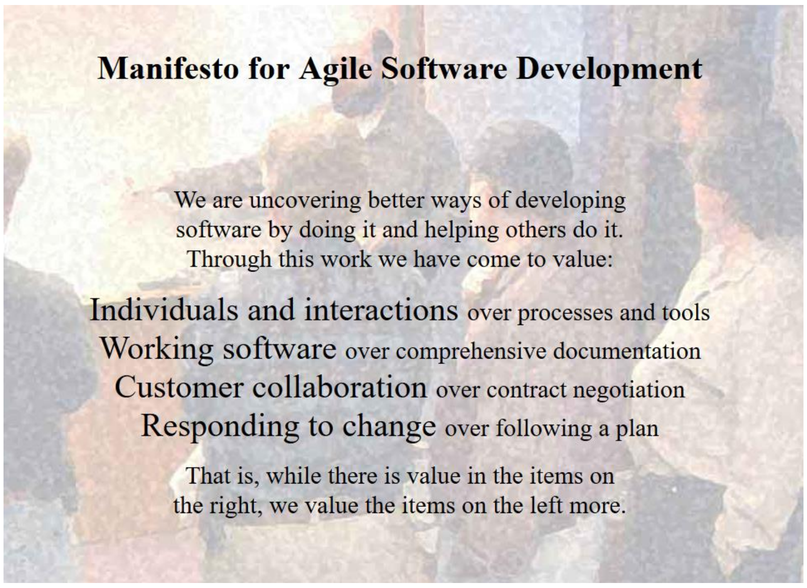
Not everything is more challenging, though. Compared to manufactured products, software is quickly duplicated and easily changed throughout releases. This offers innumerable more possibilities from a flexibility and agility perspective than in the case of manufactured products. The main message is that a direct copy/paste from manufacturing to software development is not possible. However, if Agile and Lean are thought of as a set of principles rather than practices (as presented in Section 3), applying these principles in a software domain makes more sense and can lead to process improvements (Liker, 2004; Poppendieck and Cusumano, 2012; Maglyas et al., 2012). This section develops on how, based on the principles presented in Section 3, Agile and Lean have been interpreted for software development.

4.1 Agile Software Development

The Agile Manifesto started the Agile software development movement. As Lean thinking principles, the Agile Manifesto only establishes the fundamentals of ASD. Besides, a variety of methods has emerged to implement Agile's values and principles in practice. eXtreme Programming (XP) (Beck and Andres, 2004), Scrum (Schwaber and Beedle, 2001), Dynamic Systems Development Method (Stapleton, 2003), Crystal (Cockburn, 2004) or Feature-Driven Development (Palmer and Felsing, 2001) are among the popular Agile methods. In this section, we will describe Scrum and XP as they are the most widely used in industry (Rodríguez et al., 2012), having captured also a great attention in research (see Section 2). As we will see next, whilst XP provides concrete software development practices, Scrum could be mostly considered as a software management approach.

4.1.1 Agile Values and Principles – The Agile Manifesto

Under the slogan '*We are uncovering better ways of developing software by doing it and helping others do it*', the Agile Manifesto supposed a reaction against traditional methods that were considered unable to meet market dynamics (Beck et al., 2001). Based on the original ideas of Agility, and taking Lean thinking as one of its inspiring sources (Highsmith, 2002), the Agile Manifesto was formulated from a software development angle.


Manifesto for Agile Software Development

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

Figure 5. Screenshot from the Agile Manifesto's web page (http://agilemanifesto.org/).

The Agile Manifesto highlights four values (presented in Figure 5): 1) ***individuals and interactions*** over processes and tools, 2) ***working software*** over comprehensive documentation, 3) ***customer collaboration*** over contract negotiation, and 4) ***responding to change*** over following a plan. What the authors of the Manifesto wanted to mean through these values is that while there is value in the items on the right (i.e. processes and tools, comprehensive documentation, contract negotiation and following a plan), they value the items on the left more (i.e. individuals and interactions, working software, customer collaboration and responding to change). In other words, processes and tools are useful as far as they serve people and their interactions; documentation that serves working software is fine, but documentation that is not supporting the development process and never gets updated becomes a waste of time and resources; contracts with customers are fair to run the software business, however customer collaboration is more important to develop the right product that will serve customer's needs; and being able to respond to business changes is more important than following a strict plan that might lead to create the wrong product.

In addition to these four values, the Agile Manifesto establishes twelve principles as follows:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity--the art of maximizing the amount of work not done--is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

These values and principles do not say how the software development process should be operationalized, but provide the basis to guide it. The idea is to provide *responsiveness*, *flexibility*, *capability* and *speed* to the software development process (see Agile capabilities presented in Section 3.3.). The Agile Manifesto caused some initial controversy among the Software Engineering community, especially in those who perceived Agile as an easy excuse for irresponsibility with no regard for the engineering side of the software discipline (Rakitin, 2001). However, many others understood that agility and discipline were not so opposed (Boehm, 2002; Beck and Boehm, 2003; Cobb, 2011). Indeed, the seventeen proponents of the Manifesto stated that: '*the Agile movement is not anti-methodology, in fact, many of us want to restore credibility to the word methodology. We want to restore a balance. We embrace modelling, but not in order to file some diagram in a dusty corporate repository. We embrace documentation, but not hundreds of pages of never-maintained and rarely used tomes. We plan, but recognize the limits of planning in a turbulent environment'*[8].

4.1.2 eXtreme Programming (XP)

eXtreme Programming (Beck and Andres, 2004), simply known as XP, was the most popular Agile method at the beginning of the movement. Although other Agile methods, such as Scrum, have acquired more popularity recently, many XP practices are still widely used in industry. Indeed, XP offers a very concrete set of practices to implement Agile. Overall, these practices aim to provide continuous feedback from short development cycles and incremental planning, and flexible implementation schedule to respond to changing business needs. XP relies on evolutionary design, automated tests to monitor the development process, a balance between short-term programmer's instincts and long-term business interest and oral communication.

As presented in Table 4, XP has its own set of values and principles, which are well aligned with the values and principles of the Agile Manifesto. XP's values are the basis for XP's principles (i.e. each principle embodies all values).

Table 4. XP own set of values and principles.

XP's values	1. *Communication*. Keeping a constant flow of *right* communication. 2. *Simplicity*. Finding the simplest solution that could possible work. XP argues that it is better to do it simple today and pay a little more tomorrow to change it if needed, than to do something very complicated today that may never be used in the future. 3. *Feedback*. Keeping a constant feedback about the current state of the system. Feedback works from a scale of minutes (e.g. feedback from unit testing) to a scale of days or weeks (e.g. customer feedback about the most valuable requirements - stories in XP's terminology). 4. *Courage*. Going at absolutely top speed. Tossing solutions and starting over a most promising design if needed. Courage opens opportunities for more high-risk and high-reward experiments.
XP's principles	1. *Rapid feedback*. Speed is important from a learning perspective because the time between an action and its feedback is critical to learn. XP aims to learn in seconds or minutes instead of days or week at development level, and in a matter of days or weeks at business level instead of months or years. 2. *Assume simplicity*. Each problem is faced as if it could be solved using a very simple solution. The focus is on solving today's problem and add complexity in the future, if needed. 3. *Incremental change*. Problems are solved incrementally instead of making big changes all at once. 4. *Embracing change*, by preserving the most options while solving the present problem. 5. *Quality work*. Focus on excellent or even *'insanely'* excellent to support work enjoyment.

However, what most probably took the attention of software practitioners on XP was not this set of values and principles, but its **very concrete set of practices for software development**. When creating the method, the authors tried to come back to what they argued to be the basic activities of software development: *coding* (the one artifact that development absolutely cannot live without), *testing* (and particularly automated testing, to get confidence that the system works as it should work), *listening* (so that things that have to be communicated get communicated, e.g. listening to what the customer says about the business problem) and *designing* (creating a structure that organizes the logic of the system). XP's practices focus on these four basic activities. Next, we introduce each of the twelve practices that are considered in XP's. For a more detailed description of each practice, the reader is referred to (Beck and Andres, 2004).

- *The planning game*. The planning game is used to determinate the scope of the next release. XP calls for implementing the highest priority features first. So, any features that slip past the release will be of lower value. Following the Lean principle of value, the customer is responsible to choose the smallest release that makes the

[8] http://agilemanifesto.org/history.html (last accessed, 11.08.2017)

most business value, whilst programmers are responsible for estimating and completing their own work. Particularly, two sources are used to determine the scope of a release: business priorities and technical estimates. Business stakeholders (or customer representatives) decide upon the scope of the release, priorities, composition and dates. Technical stakeholders make estimations of how long a feature takes to implement, technical consequences of business decisions, work organization (process) and detailed scheduling. The plan is an alive artefact that is quickly updated when the reality overtakes the plan. Plans are updated at least for each one- to four-week iterations and for one- to three-day tasks, so that the team can solve plan deviations even during an iteration.

- *Small releases*. XP calls for short release cycles, a few months at most. Within a release, XP uses one to four week iterations. Short releases cycles include less changes during the development of a single release and better ability to adapt to business changes. Key agile characteristics such as responsiveness and speed are achieved by working in short release cycles.
- *Metaphor*. A metaphor is a simple story of how the system should work that is shared among all programmers. The metaphor helps everyone on the project understand the basic elements of the system and their relationships. The metaphor matures as the development proceeds. It is similar to the concept of system architecture but simple enough to be easy to communicate between technical and business stakeholders.
- *Simple design*. Design should be as simple as possible to solve today's problem (e.g. runs all tests, does not have duplicated logic, has the fewest possible classes and methods). Any complexity should be removed as soon as it is discovered. Design is part of the daily business of all programmers in XP, in the midst of their coding.
- *Testing.* Testing and coding are simultaneously conducted. Developers continuously write unit tests that are run for the development to progress. Unit tests written by programmers help convince themselves that their programs work well. In addition, customers write (or at least specify) functional tests to convince themselves that the system as a whole works the way they think it should work. A comprehensive suite of tests is created and maintained, which are run and re-run after each change (several times a day). Automated testing is key in this context. Test-driven development, the concept of writing first tests for new functionality before the new functionality is added, has acquired a lot of popularity in XP (Beck, 2003).
- *Refactoring*. Refactoring focuses on restructuring the system without changing its behavior in order to improve its internal quality. Refactoring is conducted often to keep the system in shape and remove unnecessary complexity.
- *Pair programming.* Two developers programming together at one machine. XP advices for *all* production code to be written in pairs. One member of the pair focuses on writing code whilst the other think more strategically about the solution in terms of tests, complexity, etc. Pair-programming allows also knowledge sharing among team members.
- *Collective ownership*. Anyone can change anything anywhere in the system at any time. The aim is that anybody who sees an opportunity to add value can do it. Thus, everybody takes responsibility for the whole system.
- *Continuous integration*. Integrate and build the system every time that a new task is completed. As tasks are small, the system gets integrated many times per day. Continuous integration is one of the most important Agile practices nowadays, key for continuous delivery and continuous deployment (see Section 5).
- *40-hours week*. This practice aims to limit working overtime. As a rule, people should not work more than 40 hours per week so that they keep fresh, creative, careful and confident.
- *On-site customer*. XP asks for a real customer (a person who will really use the system) to be an integral part of the team, available full-time to answer questions, resolve disputes and set small-scale priorities.
- *Coding standards*. Coding standards are followed to maintain the quality of the source code. Coding standards are essential as everyone can change any part of the system at any time, and refactoring each other's code constantly. Moreover, the aim is that code serves also as a communication channel between developers.

4.1.3 Scrum

Nowadays, Scrum is among the most, if not the most, used Agile method (Rodríguez et al., 2012). Scrum was firstly mentioned as a new approach to product development in a study in Harvard Business Review in 1986 (Takeuchi and Nonaka, 1986). Thus, the ideas behind Scrum did not originate in the software domain either, but were adapted to it. Scrum uses the rugby metaphor of players packing closely together with their heads down and attempting to gain possession of the ball (see Figure 6) to refer to a holistic product development strategy where a development team works as a unit to reach a common goal. In Scrum, a common team goal is defined and shared among all team members, and all team members do their best to reach that common goal (e.g. gaining possession of the ball in rugby). The team is self-organized and can decide the best means to achieve the goal. Thus, the team leader does not *dictate* what to do next, but uses a facilitative style to help the team achieve its goal. Ideally, the team is physically co-located (reminding the concept of *cell* in Lean thinking), or uses close online collaboration, to facilitate communication and create a *team spirit*.

Figure 6. Rugby players forming a Scrum (or Melé) to gain possession of the ball.

Scrum started to be used for software development in the early 1990s. The origins of Scrum for software development is a topic of frequent debate. Some mistakenly believe that Jeff Sutherland, John Scumniotales, and Jeff McKenna invented Scrum in 1993. However, Sutherland himself has acknowledged that their first team using Scrum at Easel Corporation took Takeuchi and Nonaka (1986)'s work as their inspiring source (Sutherland, 2004). Although the team was already using an iterative and incremental approach to building software, '*the idea of building a self-empowered team where everyone had the global view of the product on a daily basis*' was motivated by Takeuchi and Nonaka (1986)'s work (Sutherland, 2004). Sutherland's team[9] started to develop what we know today as Scrum in software development. In 1995, Sutherland introduced the Scrum team to Ken Schwaber, CEO of Advanced Development Methods, who started to use Scrum at his own company as well. Sutherland and Schwaber worked together to formalize the Scrum development process, which was firstly presented in a paper describing the *Scrum methodology* at the Business Object Design and Implementation Workshop, held as part of Object-Oriented Programming, Systems, Languages & Applications '95 (OOPSLA '95). Sutherland and Schwaber, as two of the seventeen initial signatories of the Agile Manifesto, incorporated the ideas behind Scrum into it.

Scrum is a quite simple process framework. Rather than focusing on technical software development practices, **Scrum is a method for managing the software development process**. It focuses on how a software development team should organize its work to produce software in a flexible way, rather than on the technical practices that the team should apply. Figure 7 depicts the Scrum process framework. It is composed by three main element: the scrum team (in the middle of the figure), scrum events (in red) and scrum artifacts (in blue).

[9] Jeff Sutherland was the Chief Engineer (Lean role, see Table 3) for the Object Studio team at Easel Corporation that firstly started to use Scrum. The team '*defined the Scrum roles, hired the first Product Owner and ScrumMaster, developed the first Product Backlog and Sprint Backlog and built the first portfolio of products created with Scrum*' (Sutherland and Schwaber, 2011).

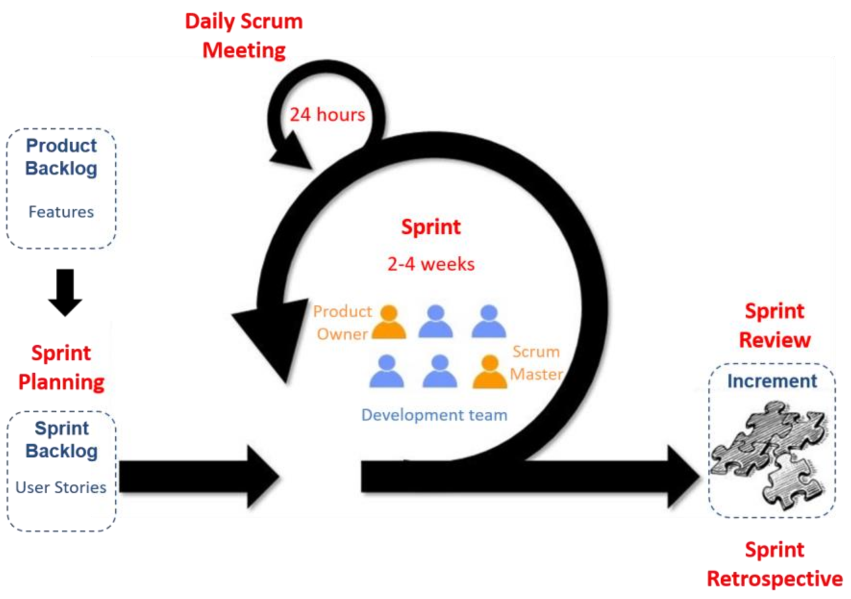


Figure 7. The Scrum process.

- The product owner, the development team and the scrum master compose the scrum team. The product owner is the responsible person for maximizing the value of the product and the work of the development team. It reminds to the Chief engineer in Lean thinking. The development team is composed by the professionals who do the work for delivering a potentially releasable increment at the end of each sprint. The scrum master is the responsible for making sure that the development work adheres to the Scrum rules (roles, events and artifacts).
- Scrum considers five well-scheduled events: the sprint, the sprint planning, the daily Scrum meeting, the sprint review and the sprint retrospective. The sprint is each of the increments in which the software development process is divided. The sprint planning is the meeting held at the beginning of each sprint to agree on the work that will be developed in the following sprint. During the sprint, the Scrum team meets daily during 15 minutes in the daily Scrum meeting to synchronize activities and create a plan for the next 24 hours. At the end of the sprint, the work done during the sprint (potentially shippable increment) is inspected during the sprint review meeting. Finally, the sprint retrospective is a meeting for the Scrum team to inspect its way of working.
- Three main artifacts are considered in Scrum: the product backlog, the sprint backlog and the increment itself. The sprint backlog is an ordered list that contains all work to be done during the sprint (e.g. user stories, features, functions, requirements, enhancements, fixes, etc.). The product backlog contains all the work to be done for the next product release. The increment is the sum of all the product backlog items completed during a sprint (i.e. outcome of the sprint).

Next, we describe how all these pieces work together. The product is defined through *product backlog items*, which are usually known as user stories and features. Product backlog items are stored initially in the product backlog and incrementally transferred to the sprint backlog as the product development progresses through sprints. User stories and features describe the product's functionality from a customer perspective. The underlying strength of these backlog items is that they focus on the value that will be delivered to the user. User stories are often defined following the pattern:

*'As a <**User**>, I want to<**Have**>so that <**Benefit**>'*

For example, an user story of a library system could be: '*As a librarian I want to have facility of searching a book by different criteria so that I will save time to serve my customers*'. In addition, each user story contains *acceptance criteria* that establish the criteria that the implementation of the user story must meet to be considered as implemented (as Done in the Scrum's jargon). User stories contain also an estimation of how much it will take to implement the story, measured as estimated *story points*. The product owner is responsible of defining the user stories and the acceptance criteria for each user story. User stories are usually described at different level of detail. Particularly, in large systems, high-level product features are initially defined and, as the work progresses, elaborated in more detail through user stories

(Leffingwell 2010). User stories and features are incrementally created. Thus, it is not expected that a complete set of backlog items is created from the beginning, but the product backlog is an alive artifact that adapts to customer/business's changes. The product owner is the customer's voice to the team and, therefore, responsible for defining features/user stories and prioritizing them to maximize the value of the product. The product owner is also responsible to ensure that the product backlog is visible, transparent and clear to everyone.

The development is carried out through increments or time-boxes, called *sprints* (lasting between 2-4 weeks). Thus, products are developed iteratively and incrementally, maximizing opportunities for feedback and, therefore, increasing responsiveness and flexibility. Each sprint has a *sprint goal* that is defined during the *sprint planning meeting*, including the user stories that will be undertaken during that sprint. This is the common goal shared by everyone in the team (as in rugby). The Scrum team is self-organised and has the authority to decide on the necessary actions to complete the sprint goal, which is defined in cooperation with the product owner. Scrum teams are cross-functional, which means that the team has all the skills and competences needed to meet the goal. Concerning the size of the team, scrum teams should be small enough to remain nimble but large enough to be able to complete significant work within a sprint. In practice, scrum teams are composed of between five to ten members. Although there are also distributed scrum teams, the ideal case is that all team members are collocated in the same working area, as the *Cell* and *Big-room* concepts in Lean thinking. Figure 8 shows an example of a Scrum team area at Ericsson R&D Finland (Rodríguez et al., 2013).

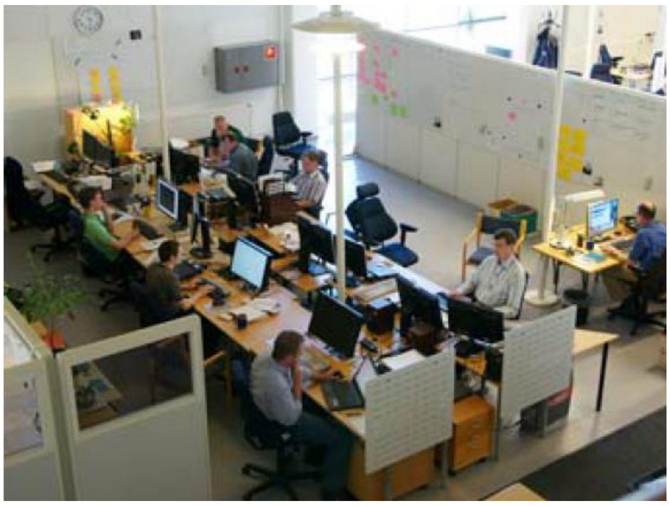


Figure 8. Scrum team area at Ericsson R&D Finland (source: Rodríguez et al., 2013).

The *sprint backlog*, selected from the *product backlog*, contains a prioritized set of user stories (or any other backlog item such as bug fixes) that will be implemented in that sprint. Once the sprint backlog is established during the sprint planning meeting, it remains unchangeable during the sprint. This means that no one is allowed to ask the development team (or any team's member) to work on a different set of requirements or tasks and, similarly, the development team is not allowed to act on what anyone else may request. After every sprint, two meetings are held. On the one hand, the *sprint review meeting*, to analyze the progress and demonstrate the current state of the product. The goal of this meeting is to show to all interested stakeholders, and particularly to customers/product owners, the potentially releasable increment that has been developed during the sprint. The key aspect in Scrum is that customer value is delivered after each sprint. On the other hand, a *retrospective meeting* is also held at the end of each spring to reflect about the way of working. In particular, the purpose of the meeting is to reflect on how the last sprint went in order to identify 1) aspects that went well and, therefore, should be kept, and 2) potential improvements that need corrections. The outcome of this meeting is an actionable plan for implementing improvements to the way the scrum team does its work. This mechanism allows self-reflection and continuous improvement (kaizen) in Scrum. In addition, every day the scrum team has a *stand-up meeting* of around 10-15 minutes, usually at the beginning of the day, in which each team member discusses what s/he did since the last stand-up meeting, whether s/he faced any obstacle in that task and what s/he will do before the next stand-up meeting in the following day. The daily scrum meeting provides transparency on the scrum team daily work and reduces complexity. It allows the team to inspect their progress towards the sprint goal, reveals possible impediments and promotes quick decision-making and adaptation.

In this process, the scrum master ensures that the rules of Scrum are followed and is in charge to remove the possible impediments in the work of the scrum team. For example, the scrum master can help the product owner by finding techniques for an effective management of the product backlog, or to the scrum team by facilitating scrum events. Therefore, the scrum master is a servant-leader for the scrum team rather than a manager.

As described above, Scrum is a fairly simple method. However, despite its simplicity, Scrum builds upon three important pillars: *transparency*, *inspection* and *adaption*. Scrum provides transparency to the development process by making progress visible all the time through events such as daily scrum meetings and sprint reviews. Scrum allows frequent inspection to detect undesirable variances (e.g. changes in customers' needs) and adapt to the new circumstances (e.g. the resulting product will not meet customer's expectations). In words of its founders, '*Scrum implements the scientific method of empiricism [...] to deal with unpredictability and solving complex problems*' (Schwaber and Sutherland, 2016). Through small iterations, Scrum improves predictability and control risk by making decisions based on what is known.

Besides the main elements of Scrum described above, other practices have adhered to Scrum during time. Although, there are not part of the original Scrum framework, they have shown to be very useful as well in certain contexts. For example, Scrum of scrums (or meta-Scrum) are used to synchronize the work of several scrum teams when scaling Scrum from one team to several teams. Backlog grooming is a popular practice to keep the product backlog in shape by reviewing that the backlog contains appropriate items that are properly prioritized. The Agile glossary by the Agile Alliance is a good resource to learn more about different terms in the context of Scrum and Agile software development in general (https://www.agilealliance.org/agile101/agile-glossary/).

4.2 Lean Software Development

Nowadays, Lean software development is certainly tied to Agile software development. However, few know that the interest of the software development industry on Lean thinking dates to early 1990s. Next, we analyze the path followed by Lean Thinking within software development before and after the Agile Manifesto, which constituted a breach in the conception of Lean thinking in software development. Then, we focus on Kanban, since it is the most popular Lean technique applied in software development, and list different sources of waste in software development.

4.2.1 Lean Software Development – pre Agile Manifesto

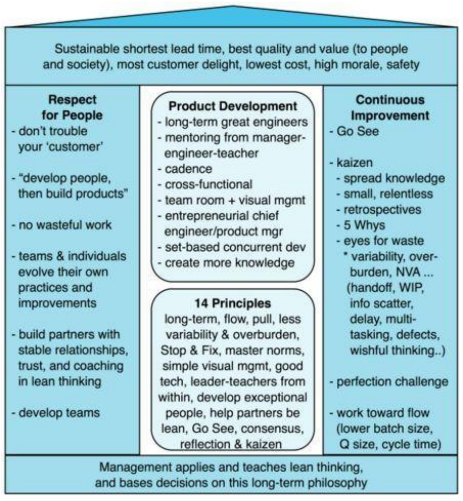
The path of Lean thinking within software development started as early as the 1990s, with concepts such as Lean software production and mistake-proofing[10] (Freeman, 1992; Tierney, 1993). At that time, Lean thinking was understood as a way of making software development processes more efficient and improving their quality. A major focus was on waste reduction. For example, Freeman (1992) describes Lean software development as a system of '*achieving ends with minimal means*' by striving to have '*the minimum amount of tools, people, procedures, and so on*'. This and similar interpretations of Lean thinking made Lean to be quite unpopular among practitioners, who saw it as a way to get rid of employees. In 1998, Raman (1998) analysed the feasibility of Lean in software development from a wider perspective, considering the five principles of *value*, *value stream*, *flow*, *pull* and *perfection* as defined by MIT's researchers (See Section 3.1). Raman concluded that Lean could be implemented through contemporary concepts (at that time), such as reusability, rapid prototyping, spiral model or object-oriented technologies. The conception of Lean software development has considerably evolved after the formulation of the Agile Manifesto.

4.2.2 Lean Software Development – post Agile Manifesto

After the formulation of the Agile Manifesto, Lean software development was mainly considered as one Agile method (Dybå and Dingsøyr, 2008). However, as practitioners started to learn more about Lean principles, it incrementally acquired an identity in itself (Wang et al., 2012). As in manufacturing, Lean software development has been differently interpreted by several authors. Table 5 presents the most popular interpretations of Lean in software development.

[10] See Table 3 for a definition of this concept.

Table 5. Interpretations of Lean thinking in software development.

Author	Description
Poppendieck and Poppendieck (2003, 2006, 2009, 2013)	Seven principles guide Lean Software Development (2006): 1. *Eliminate waste*, understanding first what value is. 2. *Build quality in*, by testing as soon as possible, automation and refactoring. 3. *Create knowledge*, through rapid feedback and continuous improvement. 4. *Defer commitment*, by maintaining options open and taking irreversible decisions when most information is available. 5. *Deliver fast*, through small batches and limiting WIP. 6. *Respect people*, the people doing the work. 7. *Optimise the whole*, implementing Lean across an entire value stream. Seven principles of Lean Software Development (2017[11]): 1. ***Optimize the whole*** as the synergy between parts is the key to the overall success. 2. ***Focus on customers*** and the problems they encounter. 3. ***Energize workers*** to bring creative and inspired people. 4. ***Reduce friction*** in terms of building the wrong product, building the product wrong and having a batch and queue mentality. 5. ***Enhance learning*** to respond to future as it unfolds. 6. ***Increase flow*** by creating a steady, even flow of work, pulled from a deep understanding of value. 7. ***Build quality in***, defects should not be tolerated and should be fixed in the moment they occur. 8. ***Keep getting better***, by changing as fast as the world changes, paying attention to small stuff and using the scientific method.
Middleton and Sutton (2005)	Interpretation based on Womack and Jones's (1996) five principles of Lean: 1. Value, identifying what really matters to the customer 2. Value stream, ensuring every activity adds customer value 3. Flow, eliminating discontinuities in the value stream 4. Pull, production is initiated by demant 5. Perfection Middleton and Sutton (2005) provides means to implement these principles in practice in a software domain.
Larman and Vodde (2009)	Larman and Vodde (2008) focuses on scaling lean and agile software development with large-scale Scrum. Their interpretation of Lean thinking based on the Lean thinking house presented below. In addition, Larman and Vodde provided a companion book with more concrete practices for scaling lean and agile (Larman and Vodde, 2010).  Sustainable shortest lead time, best quality and value (to people and society), most customer delight, lowest cost, high morale, safety Respect for People - don't trouble your 'customer' - "develop people, then build products" - no wasteful work - teams & individuals evolve their own practices and improvements - build partners with stable relationships, trust, and coaching in lean thinking - develop teams Product Development - long-term great engineers - mentoring from manager-engineer-teacher - cadence - cross-functional - team room + visual mgmt - entrepreneurial chief engineer/product mgr - set-based concurrent dev - create more knowledge 14 Principles long-term, flow, pull, less variability & overburden, Stop & Fix, master norms, simple visual mgmt, good tech, leader-teachers from within, develop exceptional people, help partners be lean, Go See, consensus, reflection & kaizen Continuous Improvement - Go See - kaizen - spread knowledge - small, relentless - retrospectives - 5 Whys - eyes for waste * variability, over-burden, NVA ... (handoff, WIP, info scatter, delay, multi-tasking, defects, wishful thinking..) - perfection challenge - work toward flow (lower batch size, Q size, cycle time) Management applies and teaches lean thinking, and bases decisions on this long-term philosophy *Summary of the Toyota Way (Lean Thinking) House by Craig Larman and Bas Vodde. 2009*
Reinertsen (2009)	Set of principles of product development flow, including managing queues, reducing batch size, applying WIP constraints, etc.
Anderson (2010)	Kanban as a means to bring Lean thinking into a software development organisation. Kanban uses five core properties: 1. visualise workflow, 2. limit WIP, 3. make policies explicit, 4. measure and manage flow, and 5. use models to recognise improvement opportunities. More details on Kanban are provided in Section 4.2.3.

[11] http://www.poppendieck.com/, last accessed 15.08.2017

Lean Software Development was initially mainstreamed with an interpretation of Lean thinking led by Mary and Tom Poppendieck (Poppendieck and Poppendieck 2003). Some argue that this interpretation took the view on software development from the Lean (manufacturing)'s angle. Still, Poppendiecks' principles have been widely used in industry. They have evolved during the years (Poppendieck and Poppendieck 2003, 2006, 2009 and 2013). Table 5 presents Poppendieck and Poppendieck original principles as they were formulated in 2006 (Poppendick and Poppendieck, 2006) and the current form of these principles. Besides some changes in terminology, the original principles have been complemented with: 1) a deeper *focus on customers* and the problems they encounter; 2) a greater focus on people and creating a Lean mindset (Poppendieck and Poppendieck, 2013); 3) *reducing friction* in terms of reducing building the wrong thing, reducing building the thing wrong and reducing a batch and queue mentality; 4) a greater focus on *enhancing learning*; and 5) a greater focus on *continuous improvement* (keep getting better).

As time passes by, more diversity was introduced. Thus, besides Poppendieck and Poppendieck's interpretation of Lean software development, other authors have contributed to shape this phenomenon. For example, Middleton and Sutton (2005) provides means to implement the original Womack and Jones's (1996) five principles of Lean thinking in the software domain. In a similar vein, Larman and Vodde (2009) base their interpretation of Lean software development on the Lean thinking house and provide concrete practices for scaling lean and agile based on this interpretation. Other authors, such as Reinertsen (2009) and Anderson (2010), have focused on more concrete aspects such as flow and Kanban. Moreover, an increasing body of scientific studies reporting experiences when using Lean software development has populated the software engineering literature (e.g. Middleton, 2001; Middleton et al., 2005; Petersen, 2010; Middleton and Joyce, 2012; Mehta et al., 2008; Staats et al., 2011; Trimble and Webster, 2013; Rodríguez et al., 2013; Rodríguez et al., 2014). Although there is no unified model for the use of Lean and Agile in these studies, some patterns are observable. Most studies converge in the importance of aspects like frequent builds through iterative development, reducing WIP, continuous problem correction and improvement, using cross-functional teams, increasing transparency and considering human aspects. Table 6 presents some benefits and challenges when applying Lean thinking in software development, as they are mentioned in these scientific studies.

Table 6. Some benefits and challenges when using Lean software development according to the scientific literature.

Benefits	**Challenges**
- Increased customer satisfaction - Increased productivity - Decreased lead and development times - Improved progress transparency - Identification of problems' root causes. - Identification of bottlenecks.	- Creating a Lean and Agile organisational culture to extend Lean thinking beyond software development teams. - A deep organizational change is needed to align the entire organization around Lean. - Involving management in development tasks. - Achieving flow in knowledge work. For example, lack of repetition in knowledge work make it difficult to specify/standardize tasks. - Frustration when starting to use the stop-the-line technique due to the need of stopping-the-line continuously.

4.2.3 Kanban

Kanban is a popular Lean technique used in software development. It is defined as a **signals system to implement the principle of pull**[12] in manufacturing. The idea behind Kanban is very simple. It focuses on using a visual signaling device to give authorization and instructions for producing items in a pull system. The signals tell an upstream process the type and quantity of products to make for a downstream process. Thus, products (or product parts) are only implemented when needed. Kanban cards are the most common example of these signals. However, Kanban can be also implemented through other devices such as metal plates or colored balls. Figure 9 shows an example of how Kanban cards support pull. Process A only produces when it receives a Kanban card. Similarly, Process B returns the Kanban card to Process A when it needs more materials.

[12] Kanban supports also other principles such as flow and value stream but its main focus was originally on pull.

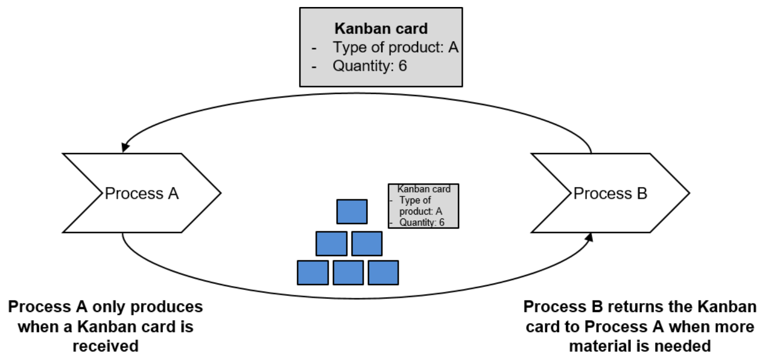

Kanban card
- Type of product: A
- Quantity: 6
Process A
Process B
Kanban card
- Type of product: A
- Quantity: 6
Process A only produces when a Kanban card is received
Process B returns the Kanban card to Process A when more material is needed

Figure 9. Kanban as a signaling device in manufacturing.

This same idea has been applied in software development but in the form of a board and sticky notes, as depicted in Figure 10. The board represents each activity in the development cycle in a column (e.g. definition of backlog items, analysis, development, acceptance testing, deployment, etc.). The sticky notes represent user stories or tasks from the product backlog. Thus, the Kanban board visualizes the development flow of these user stories/activities. The number of items under each activity is limited by establishing work-in progress (WIP) limits. For example, in Figure 10, three backlog items can be under analysis at the same time, three under development, two in the phase of acceptance testing and one in deployment. Items from the backlog are moved from column to column as the development progresses. An item can be moved to the next development step only if there is room for the new item according to the WIP limits. Thus, teams are not overloaded with work and can focus on the items that provide maximum customer value. Teams can pull more work at any time, triggered by availability of resources. For example, whenever a team member is free, s/he can pull the highest priority ticket from the product backlog. Besides development activities and WIP limits, the Kanban board can include also other information such as the person/s working on each backlog item. It is up to the team to design the Kanban board that better serves its needs.

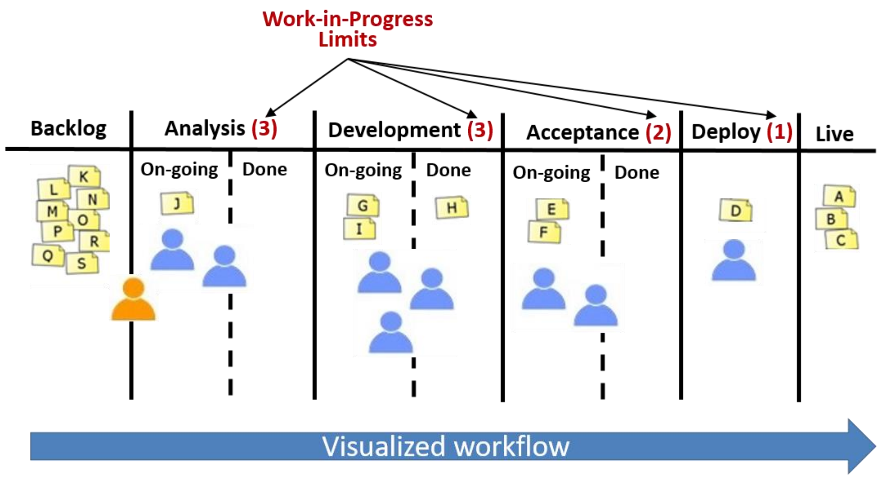


Figure 10. Example of a Kanban board for software development.

The board has the same purpose that signals in a manufacturing context, implementing pull in practice by visualizing the workflow (providing visibility to the entire software development process), and enabling the identification of problems, such as bottlenecks in the workflow (a source of waste). Moreover, WIP limits help reduce inventory and focus on the feature that is being developed (Anderson, 2009). Kanban provides five key benefits (Anderson, 2009):

1. *It visualizes workflow.* The Kanban board visualizes the status of each activity in the development process. The team can easily see the big picture of what has been done so far and what needs to be done. Transparency provides a greater control of the workload to deliver in a continuous manner.
2. *It limits work-in-progress*. WIP limits are used to limit the amount of work that can be pulled into each activity. It helps optimize workflow and prevent from team overloading and multitasking. Moreover, WIPs limits make bottlenecks visible, helping to a better distribution of resources.
3. *It helps measure and manage flow*. Besides providing information to eliminate bottlenecks in the entire value

stream, lead times can also be easily calculated using the board. The shorter the lead-time, the greater is the value for customers and the organization.

4. *It makes process policies explicit*. The process and the policies that the team agrees on how to manage the workflow becomes also visible through the Kanban board. For example, it is clear for everyone what kind of items take priority at a certain time in the development process (e.g. bug fixing versus new user story development).
5. *It supports continuous improvement collaboratively*. The Kanban board allows for transparency in the development flow and process monitoring. Inefficiencies and bottlenecks are visible. Kanban teams need to work on those if they want to maintain a steady flow (Kaizen culture).

Ahmad et al. (2013) reviewed the scientific literature on Kanban in software development and found promising benefits when using Kanban, such as better understanding of the whole process by all stakeholders, especially developers, improved team communication, and increased customer satisfaction. However, they also found some challenges, such as integrating Kanban with Agile practices, changing the organisational culture according to Lean and applying Kanban in distributed environments. Regarding integrating Kanban and Agile, Kanban is often used in combination with Scrum in which is known as Scrumban (Ladas, 2009). In Scrumban, Scrum teams employ Kanban boards to visualize their development flow. Depending on the team and the situation at hand, sprints are changed by a continuous delivery flow but other Scrum's elements such as daily scrum meetings or periodic reviews are kept (Ladas, 2009).

4.2.4 Waste in software development

The main focus of Lean software development is not on reducing costs but on creating value. Still, we list in Table 7 some typical sources of waste in software development because we think they can help better understand the concept of waste in software development processes. However, readers should not memorize these sources of waste but train their eyes to seeing waste by reflecting upon these typical sources of waste.

Table 7. Sources of waste in Lean software development.

Author	**Sources of waste**	
Poppendieck and Poppendieck (2006)	Original sources of waste in manufacturing (Ohno 1998): 1. *Overproduction:* producing more quantity of product or product's components than needed. 2. *Waiting:* waiting for a previous upstream activity to be done. 3. *Transportation*: unnecessarily moving of materials. 4. *Over-processing*: unnecessarily processing by a downstream step in the manufacturing process. 5. *Inventory:* storing of materials that are not needed for the present. 6. *Movement:* unnecessary motion of people. 7. *Defects:* rework because of quality defects.	Translation to software development (Poppendieck and Poppendieck 2006): 1. *Extra features*: producing more features and, therefore, code than what is needed to get the customer's current job done. 2. *Delays*: waiting for things to happen because of delays in starting a project, decision-making, testing, etc. 3. *Handoffs*: tacit knowledge that is left behind in the mid of the originator. 4. *Relearning*: forgetting things that were already learned so that knowledge is not preserved. 5. *Partially done work*: partially done software that is not integrated into the rest of the environment. For example, uncoded documentation, unsynchronized code, untested code, undocumented code or undeployed code. 6. *Task switching*: switching time to get into the flow of the new task. Software development requires deep concentrated thinking and switching between tasks takes a lot of effort (e.g. when people are assigned to work on multiple projects). 7. *Defects*: rework because product defects. A primary focus should be on mistake-proofing the code and making defects unusual.
Sedano et al. (2017)	1. *Building the wrong feature or product*: cost of building a feature/product that does not address user or business needs. 2. *Mismanaging the backlog*: cost of duplicating work, expediting lower value user features, or delaying necessary bug fixes. 3. *Rework*: cost of altering delivered work that should have been done correctly but was not. 4. *Unnecessarily complex solutions*: cost of creating a more complicated solution than necessary, a missed opportunity to simplify features, user interface, or code. 5. *Extraneous cognitive load*: costs of unneeded expenditure of mental energy. 6. *Psychological distress*: costs of burdening the team with unhelpful stress. 7. *Waiting/multitasking*: cost of idle time, often hidden by multi-tasking. 8. *Knowledge loss*: cost of re-acquiring information that the team once knew. 9. *Ineffective communication*: cost of incomplete, incorrect, misleading, inefficient, or absent communication.	

Poppendieck and Poppendieck (2003) translated the original seven sources of waste in Lean manufacturing (Ohno, 1998) into the seven wastes of software development. In their books, they also provide some ways to reduce these wastes. More recently, Sedano et al. (2017) provided further insights into waste in software development by empirically identifying sources of waste such as knowledge loss, ineffective communication and psychological distress.

4.2.5 Combining Agile and Lean software development in practice

The principles of Lean thinking in software development (see Table 5) are mostly well aligned with the principles and values of the Agile Manifesto. Lean inspired many ideas behind the Agile Manifesto and, therefore, it is not surprising that Agile and Lean share many similarities in the software domain (Highsmith, 2002; Conboy, 2009; Vilkki and Erdogmus, 2012). Indeed, software companies have traditionally used both in combination. The use of Lean thinking alone is quite uncommon in the software domain (Rodríguez et al., 2012). **Unlike manufacturing, the transformation towards Agile and Lean in software development has been conducted as a single trip, where the borders between Agile and Lean are not clearly defined**. Time and space restrictions, as considered in manufacturing (see Section 3.4), have not been taken into account in the software domain. Wang et al. (2012) identified six strategies that the software industry follows to combine Agile and Lean thinking. Those strategies are shown in Table 8 (shorted by popularity).

Table 8. Six categories of Lean application in Agile software development (based on Wang et al., 2012).

Strategy	Description
Using Lean principles to improve Agile processes	Selected elements or principles of Lean thinking are used in the context of Agile software development, which is already in place, to improve Agile processes. For example, automating the workflow, using the idea of eliminating waste or applying kanban.
Using Lean to facilitate Agile adoption	Lean principles serve as bases to decide the concrete Agile practices or methods to be used. Lean principles justify the adoption of certain Agile practices/methods.
Using Lean to interact with other business areas	Agile processes are kept in software development whilst Lean principles are used to interact with neighbouring business units such as delivery units, product management, marketing, customers, etc.
From Agile to Lean software development	Agile is in place when the company decides to focus on Lean. A comprehensive application of Lean principles changes Agile processes to a situation that, at the end, Lean processes become dominant and Agile practices play a supporting role.
Synchronising Agile and Lean	Agile teams and Kanban teams work in parallel and in a synchronised manner to address different development aspects of the same product. For example, Scrum teams take care of large-scale changes whilst Kanban teams focus on small feature requests and bug fixes.
Non-purposeful combination of Agile and Lean	Combination of elements from Agile and Lean with no specific distinction between both paradigms.

In our experience, **Lean thinking works well as a means for scaling Agile**, due to the unique focus of Lean thinking on the organisation as a whole (Rodríguez et al., 2013; Rodríguez et al., 2014). For example, in one of our studies with Ericsson R&D Finland, we found that Lean thinking was used to support concrete Agile practices. Ericsson R&D transformed its processes from following basic Agile principles to complementing them with Lean principles. The transformation involved around 400 people. As Figure 11 illustrates, Lean thinking underpins Agile software development. Moreover, Lean is seemed as a means to achieve a learning organization through continuous improvement. The transformation affected the whole development chain, from the earliest product release phases to maintenance, going way beyond processes and tools and involving a profound change of culture and thinking. Big projects gave room to flexible releases. Agile feature-oriented development, with an end-to-end user story focus, was used to provide flexibility in managing the product and a better ability to serve multiple stakeholders. Development was structured to supports continuous flow. Continuous Integration was a key element in this sense. Testing was done in parallel with development. As a result, the status of the software at all levels became visible and feedback times between traditional test phases were reduced from months to weeks. Seating facilities were also largely modified to support the Lean way of working. Individual offices were gradually given way to team spaces as shown in Figure 8.

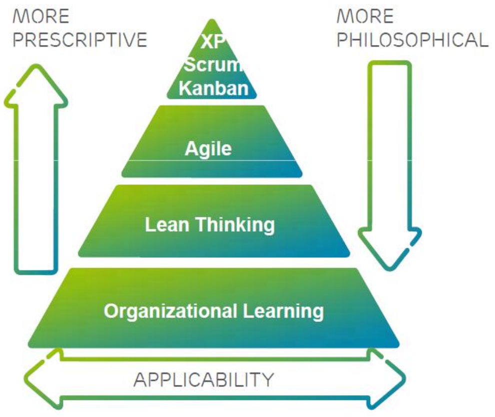


Figure 11. Lean and Agile in context at Ericsson R&D Finland (source: Rodríguez et al., 2013).

Figure 12 shows the main elements that characterize the combination of Agile and Lean in another example from Elektrobit[13], a Finnish provider of wireless embedded systems (Rodríguez et al., 2014). In general, **Lean principles guide the implementation of Agile methods**. Many elements of Agile methods such as Scrum and XP are used (e.g. network of products owners, product backlogs, continuous integration, test automations, self-organised and empowered cross-functional teams, retrospectives, etc.). In addition, Lean principles, such as creating a culture of customer value in which everyone cares about providing customer value, seeing-the-whole value stream, providing continuous flow through small batches of working software and creating a learning organization to adapt to business and market changes, guide team level activities. Lean practices, such as Kanban, are also used at implementation level.

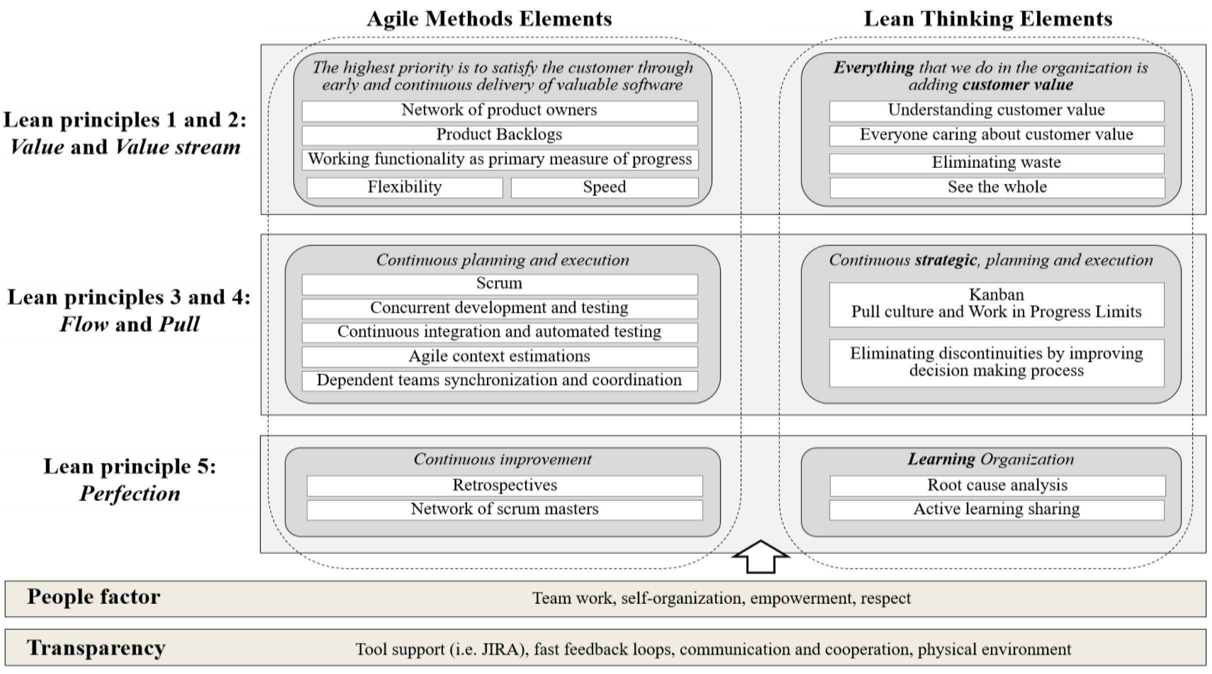

Agile Methods Elements
Lean Thinking Elements
Lean principles 1 and 2: Value and Value stream
The highest priority is to satisfy the customer through early and continuous delivery of valuable software
Network of product owners
Product Backlogs
Working functionality as primary measure of progress
Flexibility
Speed
Everything that we do in the organization is adding customer value
Understanding customer value
Everyone caring about customer value
Eliminating waste
See the whole
Lean principles 3 and 4: Flow and Pull
Continuous planning and execution
Scrum
Concurrent development and testing
Continuous integration and automated testing
Agile context estimations
Dependent teams synchronization and coordination
Continuous strategic, planning and execution
Kanban
Pull culture and Work in Progress Limits
Eliminating discontinuities by improving decision making process
Lean principle 5: Perfection
Continuous improvement
Retrospectives
Network of scrum masters
Learning Organization
Root cause analysis
Active learning sharing
People factor
Team work, self-organization, empowerment, respect
Transparency
Tool support (i.e. JIRA), fast feedback loops, communication and cooperation, physical environment

Figure 12. Main elements characterizing EB Wireless Segment's Agile and Lean way of working (source: Rodríguez et al., 2014).

Overall, **Lean helps improve Agile processes and support well-known management issues**, such as shorting release cycles and decreasing time-to-market by using flow, defining key performance indicators based on value, improving collaboration with customers by using pull and avoiding short term thinking by using perfection and continuous improvement (Maglyas et al., 2012).

[13] Elektrobit changes its name to Bittium in 2015.

5. Beyond Agile and Lean: Towards Rapid Software Development, Continuous Delivery and Continuous Deployment

Speed has become increasingly important in current software development. Indeed, it is the key term in the latest advances beyond Agile and Lean software development processes in the form of rapid software development. **Rapid software development refers to *develop, release* and *learn* from software in rapid parallel cycles, often as short as hours or days** (Mäntylä et al., 2015; Fitzgerald and Stol, 2017). In this context, software is not seen anymore as an item that evolves through releases that are delivered every certain months; rather, software is seen as a continuous evolution of product functionality that is *quickly* delivered to customers. Value-driven and adaptive real-time business paradigm are key concepts in this context (Järvinen et al., 2014). Value-driven because the goal is to create as much value as possible for both, the company and its customers. Adaptive real-time business paradigm because that value is created by activating a continuous - real time- learning capability that allows companies to adapt to business dynamics. In this section, we develop into the importance of speed in current software development processes and describe the concepts of continuous delivery and continuous deployment as a means to achieve rapid software development. Particularly, we focus on process elements that are important to achieve this capability.

5.1 Need for speed

Why does speed matter? The traditional view of software as a static item, which can be bought and owned, is not the current reality of the software industry. Software products and software-intensive systems are increasingly object of a constant transformation. Indeed, customers expect a continuous evolution of product functionality that provides additional value (Bosch, 2012). Speed matters because it provides a competitive advantage to satisfy customers' needs. Speed is concerned with shortening feedback loops between a company and the users of its products. It allows companies to better understand user's behaviour and adapt to business/customer changes faster. **The risk of predicting what customers want is decreased by increasing feedback loops between users and companies and, thus, *learning* (instead of predicting) what customers want**. Need-for-Speed (N4S) is the name of the research program in which we have conducted part of our research activities during the last few years (http://www.n4s.fi/en/). Executed by the forefront Finnish software companies and research institutions, N4S focused on making software business in real time. Concretely, during the program, we investigated solutions for adopting a real-time experimental business model to provide capability for instant value delivery, based upon deep customer insight. In the rest of the chapter, we reveal our most relevant findings from the program. Particularly, we discuss key process elements that are important to get speed in software development and the concept of lean start-up as an experimental model to foster innovation.

Coming back to the reasons why speed is important in software development processes, in his recent book *Speed, data, and ecosystems: excelling in a software-driven world* (Bosch, 2017), Bosch reflects on five reasons why speed matters in current software business. According to Bosch,

1. *Speed increases the company's ability to adapt development priorities*. Working in short increments (such as sprints) allows companies set-up priorities more often. Obviously, the overall product's direction cannot be changed every increment but priorities in small items such as features can be adapted.
2. *Speed allows accelerating the innovation cycle by a faster iteration through the phases of build-measure-learn (BML)*. Speed allows companies learn what new ideas would work for their customers faster than the competence. We further develop the concept of BML and the Lean startup movement, in which BML originated, in Section 7.
3. *Speed gives development teams confidence to go into the right direction as it enables believable progress tracking*. Team satisfaction increases as team's members realize that they are able to provide customer value on a continuous basis.
4. *Speed allows continuous feedback on the delivery of value.* It enables fast user feedback to guide the following development steps. Data collection is fundamental in this phase so that decisions are based on data rather than intuition or subjective opinions.
5. *Speed supports product quality.* Faster and more frequent release cycles should not compromise software quality. In fact, although it may seem paradoxical, when properly applied, rapid software development favours quality. In rapid software development coding and testing are done in parallel. Integration and bug fixing are not left to

the end of the process, and testing is highly automatized. Overall, speed forces organizations to maintain high quality levels all the time, as new software is continuously delivered to users.

As we described in Section 3.3, speed is one of the four main capabilities of Agility. Speed is embedded in the Agile Manifesto as well. For example, Agile principles, such as '*our highest priority is to satisfy the customer through early and continuous delivery of valuable software*' and '*deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale*' make clear reference to speed. The notion of sprints in Agile methods such as Scrum aims also to increase speed in the software development process. Instead of waiting until the end of development, customers can see progress after every sprint. Based on Agile software development principles, rapid software development increases speed to approaches where the step between development and deployment is minimized and code is immediately delivered to production environment for customers to use (and learn from the customer's usage of the product). Iterative Agile and Lean software development are extended towards continuous flow of product functionality.

5.2 Continuous delivery and continuous deployment

Continuous delivery and continuous deployment are at the heart of rapid software development. Intuitively, both relate to produce new fully evaluated and ready-to-deploy software revisions continuously, even many times per day. Humble and Farley (2010), in their book *Continuous Delivery: Reliable Software Releases through Build, Test, and deployment automation* published in 2010, state that continuous delivery provides enterprises with the ability to deliver rapidly, reliably and repeatedly value to customers at low risk with minimal manual overhead. The central concept in Humble and Farley's approach is a deployment pipeline that establishes an automated end-to-end process to ensure that the system works at technical level, executes automated acceptance tests and, lastly, deploys to a production or staging environment. The goal of continuous delivery is to enable the delivery of new software functionality at the touch of a button, using a fully automated process. In the ideal scenario, there are no obstacles to deploying to production; there are no integrations problems, builds can be deployed to users at the push of a button.

There is certain confusion in the use of these terms- continuous delivery and continuous deployment. The early scientific literature tended to use both interchangeably, as synonyms (Rodríguez et al., 2017). However, there are important differences between them. Humble and Farley (2010) describe continuous deployment as the automatic deployment of every change to production, whilst continuous delivery is an organizational capability that ensures that every change can be deployed to production, if it is targeted. However, the organization may choose not to do so, usually due to business reasons. In other words, continuous delivery refers to the organizational capability to kept software always releasable. That is, software is ready to be released to its users at any time but the company decides when it will be released. On the other hand, in continuous deployment, software is automatically deployed to production when it is ready. Note that while continuous deployment implies continuous delivery, the converse is not true. Figure 13 graphically illustrates the difference between continuous delivery and continuous deployment.

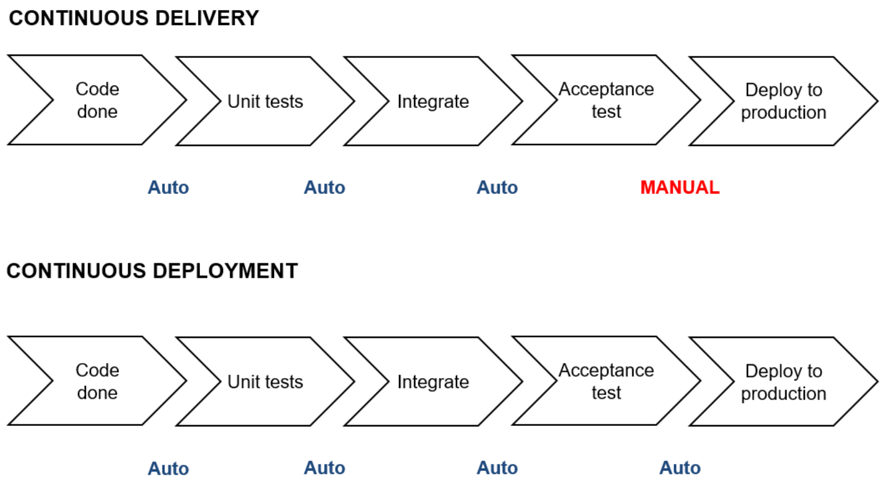


Figure 13. Conceptual difference between continuous delivery and continuous deployment.

Humble and Farley argue that although continuous deployment enables releasing every good build to users, it does not always make sense. In continuous delivery, any build could *potentially* be released to users, but it is the business, not IT, who decides about release schedules.

Leading organizations, such as Facebook, Microsoft and IBM have actively moved towards rapid releases (Claps et al., 2015). A plethora of evidence related to continuous delivery and deployment exists in companies' white papers and online blogs, where practitioners have voiced their experiences, expectations and challenges in moving to this direction (e.g. Facebook[14], Microsoft[15], Google[16], Adobe[17] and Tesla[18]). Next section develops into the main characteristics that allow companies to increase speed in their software development processes and achieve continuous delivery/deployment.

5.3 Key elements to get speed in your software development process

In the context of the Need-for-Speed project[19], we reviewed the scientific literature on continuous deployment and continuous delivery using the systematic literature review method (Rodríguez et al., 2017). Our objective was to determine the underlying factors that characterize this phenomenon as they appear in the scientific literature. Next, we describe the most relevant factors that we found in our study.

1. ***Fast and frequent releases***, with preference given to shorter cycles or even continuous flow (weekly or daily deliveries). Release cycles are accelerated when moving towards rapid software development. This implies that the surrounding areas that enable fast releases are also transformed. For example, planning needs to be a continuous activity as well in order to enable a steady flow of small changes. A tighter integration between planning and execution is needed to ensure alignment between the needs of the business context and software development. The ability to release quickly does not mean that development should be rushed into without a conscious understanding of what is actually being done. The release process needs to be clear, including having a clear release management workflow and delivery workflow.
2. ***Flexible product design and architecture***. The software architecture evolves during the product life-cycle and, therefore, needs to be able to adapt to changing conditions. The architecture must be flexible. Still, it has to be robust to allow organizations to invest resources in offensive initiatives (e.g. new functionality, product enhancements and innovation) rather than defensive efforts (e.g. bug fixes). The architecture and design have to be highly modular and loosely coupled (MacCormack, 2001; Olsson et al., 2013; Bellomo et al., 2013b). Moreover, measuring and monitoring source code and architectural quality is important (Bellomo et al. 2013a; 2013b). The literature provides some techniques to manage architecture aspects in rapid software development. For example, rapid architecture tradeoff analysis to accommodate rapid feedback and evaluate design options (Bellomo et al., 2013a), quantifying architectural dependencies by combining Design Structure Matrix (DSM) and Domain Mapping Matrix (DMM) (Brown et al., 2013) and identifying and assessing risky source code areas based on diverse metrics (Antinyan et al., 2014). In addition, in order to enable continuous and rapid experimentation (see fourth factor), mechanisms in the software architecture for run-time variation of functionality and data collection as well as rollback mechanisms to revert changes are required. The literature offers some solutions to achieve this end. For instance, Rally Software suggests A/B testing with Feature Toggle as a technique to manage and support run-time variability of functionality (Neely and Stolt, 2013).
3. ***Continuous testing and quality assurance***. Testing and quality assurance practices have to be performed throughout the whole development process, as new functionality is rolled out, and not just at the end of the development. Transparency on the testing process is important to avoid problems such as duplicated test effort and slow feedback loops (Nilsson et al., 2014). Well-known testing techniques in Agile software development, such as test automation and continuous integration support continuous quality assurance. The concept of technical debt, which is further developed in Section 8, is also important to balance speed and stability. However, besides technical aspects, continuous testing and quality assurance require also a culture of quality. Developers

[14] https://www.facebook.com/notes/facebook-engineering/ship- early- and- ship- twice- as- often/10150985860363920 (last accesed 27.06.2107)

[15] http://blogs.msdn.com/b/bharry/archive/2012/06/07/announcing-continuous-deployment- to- azure- with- team- foundation- service.aspx

[16] https://air.mozilla.org/continuous-delivery-at-google/

[17] http://steveblank.com/2014/01/06/15756/

[18] http://steveblank.com/2014/01/06/15756/

[19] http://www.n4s.fi/en/

bear the responsibility for writing good code, perform thorough tests as well as support the operational use of their software (Feitelson et al., 2013; Marschall, 2007; Trimble and Webster, 2013). As systems become larger and more complex, such culture complements test automation and allows quality to be maintained at scale. The principle is that more frequent release cycles should not compromise software quality.

The latest advances towards managing quality in rapid software development focus on supporting quality-awareness during rapid cycles through continuous monitoring and visualization of product quality. For example, our work on the Q-Rapids project[20] focuses on a data-driven, quality-aware rapid software development framework as depicted in Figure 14. In this framework, quality and functional requirements (e.g. user stories or features) are managed together (Guzmán et al., 2017). Thus, quality requirements are incrementally elicited and refined based on data gathered at both development time and runtime. Using data-mining techniques, project, development, runtime and system usage data is aggregated into quality-related strategic indicators to support decision makers in steering future development cycles. A strategic dashboard aggregates the collected data into strategic indicators related to factors such as time to market, team productivity, customer satisfaction, and overall quality to help decision makers making data-driven, requirements-related decisions in rapid cycles (Franch et al., 2017).

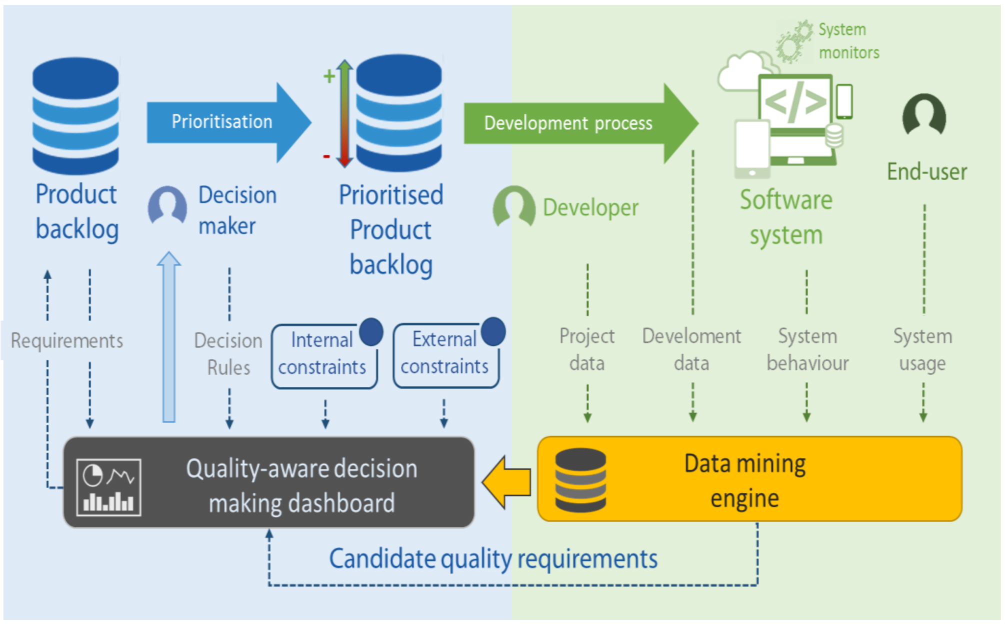


Figure 14. The Q-Rapids framework (source Guzmán et al., 2017).

4. ***Rapid and continuous experimentation***. In rapid software development, systematic design and execution of small field experiments guides product development and accelerates innovation. Continuous experimentation allows companies to learn customer needs and base business and design decisions on data rather than on stakeholder opinions (Eklund and Bosch, 2012). For example, Facebook uses *A/B (split) testing* as an experimental approach to identify user needs (Feitelson et al., 2013). In A/B testing, randomized experiments are conducted over two options (e.g. two similar features) to compare, through statistical hypothesis testing, how end-users perceive each feature (Feitelson et al., 2013; Benefield, 2009). The experimental results are continuously linked with the product roadmap in order to provide guidance for planning activities (Fagerholm et al., 2014). Rapid experiments are at the heart of the BML innovation cycle (see Section 7).
5. ***User involvement***. The role of users is essential in rapid software development. Everything goes around providing value to users. Thus, users are continuously monitored to learn what will provide value to them.

[20] Project funded from the European Union's Horizon 2020 research and innovation programme under grant agreement No 732253 (2016-2019), http://q-rapids.eu/

Mechanisms to involve users in the development process and collect user feedback from deliveries as early as possible (even near real-time) are essential in rapid software development. They guide design decisions and innovation. However, techniques need to be non-intrusive so that users are not stressed with continuous feedback requests. Several options can be use in this sense. For instance, similar to Facebook, Rally Software (Neely and Stolt, 2013) uses also A/B testing with Feature Toggle as a technique to manage and support run-time variability of functionality. Facebook uses a fast upgrade of database systems to deploy experimental software builds on a large scale of machines without degrading the system uptime and availability (Goel et al., 2014). Besides involving customers in experiments, customers are also involved in testing activities. Again, strategies applied with such purpose have to be as non-intrusive. For example, *dark deployment* is used to deliver new features or services, which are invisible to customers and have no impact on the running system, to test system quality attributes and examine them under simulated workload in a real production environment (Feitelson et al., 2013; Neely and Stolt, 2013). Another relevant practice, *canary deployment*, delivers a new version of the system to a limited user population to test it under real production traffic and use. The new version is then delivered to the whole user population once it reaches a high enough quality level (Feitelson et al., 2013; Neely and Stolt, 2013).

6. ***Automation***. Computers perform repetitive tasks, people solve problems. In the context of continuous delivery and deployment, the focus is on automating the entire delivery pipeline as much as possible. The goal is that deployments are predictable, routine tasks that can be performed on demand in a matter of seconds or minutes. Continuous integration automates tasks such as compiling code, running unit and acceptance tests, monitoring and validating code coverage, checking compliance with coding standards, static code analysis and automatic code review. The continuous integration practice of Agile software development is extended to release and deploy automation. For example, Facebook (Feitelson et al., 2013) has a tool for deployment called Gatekeeper that allows developers to turn features on and off in the code, and to select which user groups see which features. Similarly, Agarwal (2011) presents an automation system to integrate configuration management, build, release and testing processes. The system enables migrating changes from one environment to another (e.g. development to production), performing a release with selected content, and upgrading software in a target environment. Humble et al. (2006) recommend automating build, testing, and deployment in the early stage of the project and evolving the automation along with the application.
7. ***DevOps***. DevOps is another key element in continuous delivery and deployment because it allows achieving rapid end-to-end software development. DevOps integrates development and operations. It enables transparency and understanding of the whole development pipeline to overcome corporate constraints that often cause delays in product deliveries (e.g., handover delays and communication gaps). The concept of DevOps is further developed in the next section.

6. DevOps

When moving beyond Agile and lean software development towards continuous deployment, the software development process becomes characterised by increased collaboration of cross-organisational functions and automation, particularly, in software testing and deployment. In this section, we introduce the concept of DevOps, whose core principles center around cross-discipline collaboration, automation and the use of Agile principles to manage infrastructure and its configuration. It is worth noticing that, in DevOps, much focus is given to the mechanisms used to manage development environments in addition to the development of software features.

6.1. The origins of DevOps

Compared to early years of its inception, the concept of DevOps today is less ambiguous owing to the increased number of scientific studies describing its underlying characteristics and its adoption in software-intensive organisations (Lwakatare, Kuvaja and Oivo, 2015; Bass, Weber and Zhu, 2015; Callanan and Spillane, 2016; Penners and Dyck, 2015; Smeds, Nybom and Porres, 2015). DevOps emerged in 2009 as an approach for implementing continuous deployment (Humble and Molesky, 2011). For several years, it had remained unclear what the concept entails, despite its popularity in the software industry.

The term ‘DevOps’ is a combination of two words, *‘Development’* and *‘Operations’*. As a concept, **DevOps stresses on building an Agile relationship and collaboration between software development and operations**, whereby operational requirements are considered early into, and throughout the software development process. The main aim of DevOps is to speed up the delivery of quality software changes more reliably and repeatedly in production environment. DevOps proposes some practices and the use of tools to automate the management of software infrastructures, which over the years have become complex, heterogeneous and of large-scale.

Historically, the term DevOps was coined by Patrick Debois when organising a practitioners’ event called DevOpsDays[21]. Prior to DevOpsDays, in 2008, Debois gave a presentation at the 2008 Agile conference titled: ‘*Agile infrastructure and operations*’ (Debois, 2008) that focuses on employing Agile principles to infrastructure-related activities and projects, such as upgrading application server that has multiple infrastructural component dependencies. After his presentation, Debois observed a growing interest on the topic amongst other software practitioners. For instance, few months prior to the DevOpsDays event, two other pioneers of DevOps, John Allspaw and Paul Hammond, shared their experiences of multiple deployments in a presentation titled ‘*10 deploys per day: Dev & ops cooperation at Flickr*’ at the 2009 Velocity conference. Together with other software practitioners, the pioneers introduced a DevOps movement that today embodies a diverse set of practices and tools crucial for speeding up the delivery process of software changes to end-users.

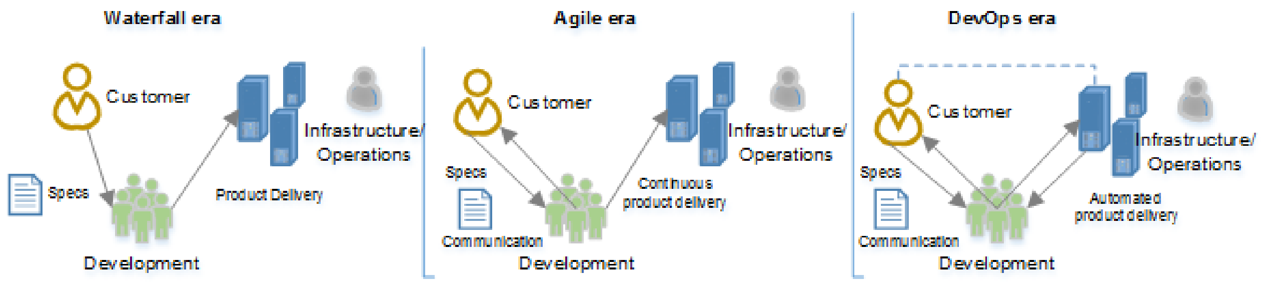


Figure 15. DevOps extending agile collaboration to operations engineers (adapted from Rautonen, 2013).

Much of what is addressed by DevOps movement deals with problems resulting from organisational silos between software development and operations. For many medium and large organisations, software development and operations activities are traditionally performed by separate organisations. Typically in such context, software is designed, implemented and tested by software developers and testers, and, upon completion, the developed software is handed over to operations personnel, such as system administrators. Operations personnel then deploy and operate software in production environment. Such a setup is beneficial to facilitate required specialisation and prevent mix up of software development and operations tasks in daily work of engineers. However, the separation, exacerbated by poor communication between software development and operations, has been reported to bring several serious problems, including *unsatisfactory testing environments and systems being put into production before they are complete* (Iden et al., 2011). The problems intensified when software developers started to use Agile methods, but without their adoption amongst operations personnel (Elbanna & Sarker, 2016; Gotel & Leip, 2007). As a result, conflicting goals of ‘*stability vs. change*’ began to appear. Operations personnel, being skeptical of the impacts of frequent changes to production environment, tend to seek stability by avoiding frequent system changes to production, which software developers tend to seek with Agile methods. To facilitate effective collaboration between software development and operations, several approaches, ranging from ad-hoc and informal to formal approaches, were suggested (Tessem and Iden, 2008). However, more recently DevOps emerged as a solution that focuses on automating the deployment pipeline and extending Agile methods to operations, as visualised in Figure 15 (Bass, weber and Zhu, 2015; Lwakatare, Kuvaja and Oivo, 2016). The deployment pipeline, visualised in Figure 16, is the path taken by all software changes to production, including changes of software features and configuration parameters. In the deployment pipeline, though it may manifest differently across software-intensive organisations, similar patterns can be observed. A common pattern is that, the software developer performs development tasks on his/her environment and may performs some tests before deciding to commit the changes to team’s development environment, where the changes are made visible to others. Upon commit, software changes are tested for integration and with successful tests a build is created. The newly created software build is then passed for

[21] DevOps Days is a worldwide series of technical conference covering topics of software development and IT infrastructure operations (http://devopsdays.org/about/).

additional tests such as acceptance tests before being deployed to production environment, where software changes are made visible to end-users. Specific details in the level and efforts needed for test and deployment automation vary across software-intensive organisations depending on the domain and type of application being developed.

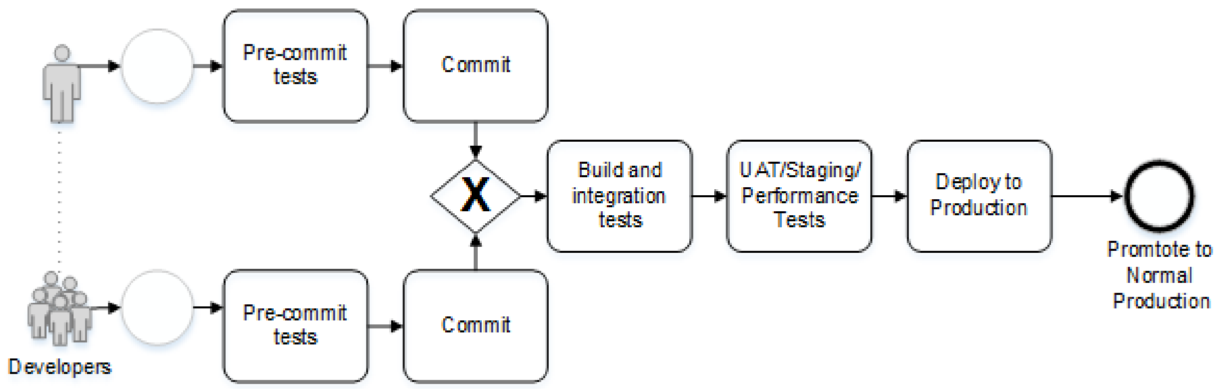

Pre-commit tests
Commit
Build and integration tests
UAT/Staging/ Performance Tests
Deploy to Production
Promtote to Normal Production
Pre-commit tests
Commit
Developers

Figure 16. The deployment pipeline (adapted from Bass, Weber and Zhu, 2015)

6.2. DevOps practices

The adoption of DevOps in software-intensive organisations is not homogenous due to factors such as objective and starting baseline. DevOps adoption entails a series of organisational and socio-technical practices to be employed. The organisational aspect focuses on organisational structures and ways in which an organisation deals with the relationships and responsibilities between software development and operations. Meanwhile, the socio-technical aspect focuses engineering practices on development practices of software changes, deployment pipeline and interaction of software developers and operations personnel with different artefacts.

From an organisational perspective, DevOps has four common practices, visualised in figure 17 (Cito et al., 2015; Shahin et al., 2017):

(a) separating development and operation with high collaboration;
(b) separating development and operation with a facilitator between the two groups;
(c) development team gaining more responsibilities of operations and operations existing in a small team and;
(d) no visible operations team.

Amongst the four practices, the practice of separating software development and operations with high collaboration is most common in large software-intensive organisations (Cito et al., 2015; Shahin et al., 2017). The practice of separating software development and operations with a facilitator between them, and assigning more responsibilities to developers are more common in small and medium sized organisations (Cito et al., 2015; Shahin et al., 2017). However, regardless of the organisation size, the practice of having developers who gain more responsibilities of operations is observed to increasingly becoming common (Cito et al., 2015; Shahin et al., 2017). This has huge implications to the working culture and division of responsibilities (Callanan & Spillane, 2016; Feitelson et al., 2013; Schneider, 2016; Smeds et al., 2015). A complimentary practice consists of establishing of a separate team that acts as a facilitator between the two groups. (Callanan & Spillane, 2016; Chen, 2017; Elberzhager et al., 2017; Schneider, 2016) argue for such a team because it can significantly reduce required efforts and minimise disruptions for development team. Particularly, in large organisations, the established team consists of a small number of members with multidisciplinary background including that of software development and operations. The team is responsible for standardising deployment mechanisms such as building a deployment pipeline platform, which is to be used by software developers (Callanan & Spillane, 2016; Chen, 2017; Elberzhager et al., 2017). An important aspect to notice is that when knowledgeable software developers are given the responsibility to perform operations tasks, they tend to be given access to critical environments (Nybom et al., 2016; Schneider, 2016). Such access allows software developers to provision by themselves different software development environments.

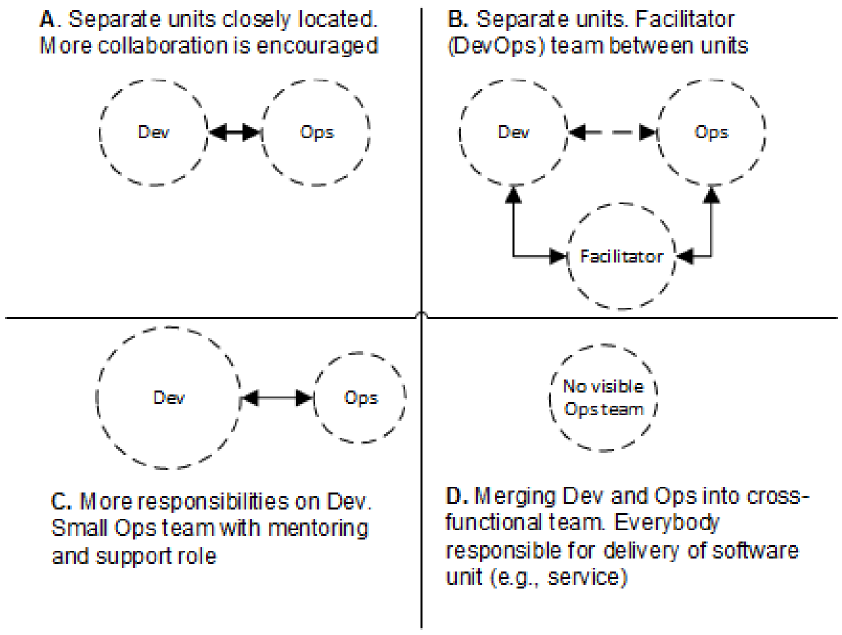

A. Separate units closely located. More collaboration is encouraged
Dev
Ops
B. Separate units. Facilitator (DevOps) team between units
Dev
Ops
Facilitator
Dev
Ops
C. More responsibilities on Dev. Small Ops team with mentoring and support role
No visible Ops team
D. Merging Dev and Ops into cross-functional team. Everybody responsible for delivery of software unit (e.g., service)

Figure 17. DevOps practices from an organizational perspective (adapted from Shahin et al., 2017).

DevOps practices from a socio-technical perspective focus on automating the deployment pipeline, including automatic management of software infrastructure consisting of development and production environments. The socio-technical DevOps practices include technical infrastructure where software development and operations activities take place and its interaction with human agents. Therefore, the deployment pipeline is the technical infrastructure where socio-technical DevOps practices are observed. In specific, a survey of software practitioners by (Cito et al., 2015) reports on incorporation of practices in the following areas for DevOps: *(1) software deployment and provisioning; (2) software design and implementation; (3) software troubleshooting; (4) software monitoring and production data usage.* Automation practices in the deployment process, which focus on automatic provisioning of environments, are more prominent in cloud-native software (Cito et al., 2015). One reason is the progress in virtualisation technology, which is made prevalent through cloud computing. Cloud computing provides access to computing resources that, through DevOps tools such as Puppet and Chef, software developers can use to provision and manage environment repeatedly and fast. The tools advocated in DevOps, which are used to automatically provision and manage environments, apply the concept of infrastructure-*as-code*, which is central to DevOps. The *Infrastructure as Code* concept entails a practice of using software engineering practices for managing source code such as programming language and version control, to implement and manage scripts used for environment provisioning, configuration management and system deployment (Hummer et al, 2013). In practice, the *infrastructure-as-code* concept is facilitated by DevOps tools like Puppet and Chef, which feature a domain-specific language used to write scripts for environment provisioning and software deployment. The benefit of using *infrastructure-as-code* is that it helps minimise inconsistencies and complexity of software infrastructure, which in turn helps shorten the release cycle. We further elaborate on two socio-technical DevOps practices as these were the focus areas in our research.

Automation of deployment process. The automation of deployment process in DevOps involves two main activities. First, an automatic provisioning of development environments e.g., test, that are similar to the production environment. Second, the application of deployment strategies such as blue/green deployment, rolling upgrade and canary deployment, to deploy software changes to desired environments. The latter is meant to replace a manual process for deploying software and configuring environment as well as to minimize environment inconsistencies with production, as these constitute to common antipatterns in software delivery process and have impacts to the release cycle time of changes (Humble & Farley, 2010). Additionally, a deployment strategy is selected and facilitated by the use of deployment automation tools. Blue/green deployment and rolling upgrade are two commonly used deployment strategies for cloud application (Bass et al., 2015). In blue/green deployment, a number of virtual machine (VM) instances, which contain old version of the software, are maintained while in parallel the same number of VM instances, which contain an updated version of the software, are provisioned and only when they are ready, traffic is routed to the new instances (Bass et al., 2015). In rolling upgrade deployment strategy, a small number of VM with new software version are upgraded directly to environment containing VM instances with old version and in parallel a number of old VM instances are switched off (Bass et al., 2015). However, each strategy may also pose some reliability issues, including the amount of time needed to prepare and place a new VM to the desired environment (Bass et al., 2015). Recently, microservice architecture has been found beneficial when considering deployment automation. Microservice architecture loosely decouples the software into collaborating services whereby each service implements a set of related functions that can be easily developed and frequently deployed, using technology such as Docker, independent of other services (Balalaie et al., 2016). Despite its many benefits, microservice architecture still presents several challenges that need to be addressed in research, including complexities introduced in managing and monitoring system comprised of many different service types (Heinrich et al., 2017).

Monitoring. In DevOps, monitoring becomes important not just for operations personnel but also software developers. For instance at Facebook, developers in a team are responsible for monitoring and teams schedule. They rotate amongst themselves an 'oncall' person who receives alerts of urgent issues through automated phone calls but also shields others to focus on development work (Tang et al., 2015). Software monitoring tools such as New Relic allow developers to install monitoring agents to the software in order to gather real-time software and server monitoring data and metrics specified as relevant by the development team (Bass et al., 2015). Insights from monitoring data can be used to inform and help developers identify software failures and performance improvements as well as gain feedback from users based on software usage amongst other things (Bass et al., 2015). Recent research contributions on this aspect have focused on finding effective ways to correlate runtime data to software changes during development time for such purpose, as to automate troubleshooting activity amongst others (Cito et al, 2017). This is contrast to the traditional approach whereby developers and operators employ manual steps to diagnosis faults by navigating and inspecting relevant information from log statements, software metrics, and using mental models (Cito et al, 2017). For this purpose, Cito et al. (2017) have proposed an analytical approach to established explicit links that carry semantic relations between related fragments that can be individual logs, metrics, or excerpts of source code.

Finally, it is relevant to mention that DevOps practices are championed and well established in software-intensive organisations that are involved in cloud and web development (Callanan & Spillane, 2016; Cito et al., 2015; Elberzhager et al., 2017; Feitelson et al., 2013). DevOps practices are increasingly being evaluated for their feasibility in other contexts, such as embedded systems, particularly because several factors make the adoption of DevOps in such contexts challenging e.g. unavailability of technology to enable flexible and automatic system redeployment repeatedly (Laukkarinen et al., 2017; Lwakatare et al., 2016; Schneider, 2016).

6.3. The benefits and challenges of DevOps

Benefits and challenges of DevOps have been reported in empirical studies. For example, among the most popular benefits are *a reduction in release cycle time; improved quality and eased communication between software development and operations* (Callanan & Spillane, 2016; Chen, 2017; Elberzhager et al., 2017). However, successful adoption of DevOps requires a mindset change, management support and shared responsibility amongst all stakeholders involved in the delivery of software. Support from senior management strengthens the value of and the adoption of DevOps across an organisation (Jones, Noppen, & Lettice, 2016). However, the organisation may encounter several challenges during the adoption of DevOps, including the need to re-architect the software to better support DevOps in deployability, testability and loggibility (Shahin, 2015). Microservice architecture is one architectural design having such benefits as scalability, flexibility, and portability, and is in support of DevOps (Balalaie et al., 2016). Furthermore, since in DevOps, the development team is given the responsibility to develop and operate the system that they develop, this requires the team to be equipped with necessary skills, thus, increasing the learning curve amongst development team members (Nybom, Smeds, & Porres, 2016). Alternatively, for some processes, their inner working may be abstracted through automation achieved in the deployment pipeline, thus, demanding a low learning curve from development team members.

7. The Lean Startup Movement

Applying the principles of Lean thinking has been broadened from the areas described in the earlier sections. One new area is the emergence and early stages of new enterprises, startups. Lean thinking is widely applicable in many areas where something new is built - manufacturing, software development, or startups. In this section, applying Lean thinking on software startups is described.

7.1 Origin and key concepts

In 2011 Eric Ries (Ries, 2011) applied the principles of Lean thinking into startup companies in his book '*The lean startup: How today's entrepreneurs use continuous innovation to create radically successful businesses*'. In the book, he focused on the question how to tune a successful business case from a product idea. In his thinking, leanness was crystallized by defining waste as a product without market potential: the biggest waste is to create a product that nobody is willing to buy. Ries presented how to work in a Lean way and avoid waste by presenting a process of early learning. The key characteristics of that process are 1) earliness, 2) close customer co-operation, 3) rapid progress, 4) measurement of the value, 5) learning, 6) corrective actions, and 7) iteration.

- ***Earliness*** refers to the need to follow lean thinking from the very beginning of a startup. In the opposite case, the course of the company may already be fixed towards a wrong direction, and the efforts done turn out to be waste.
- ***Close customer co-operation*** refers to the principle that the customer(s) need to be linked to the startup's doings as early and closely as possible. The measurement of the value of the product is fully based on the customers' opinions. A real customer is the source of the feedback needed when measuring the business potential of the innovation. Ries' book classifies customers by defining different groups of customers being relevant for a startup in different phases of its evolution path. The customer groups are defined mostly in the context of American business environment and products aimed to mass markets. In a practical situation, however, close customer relationship is easier to create in case of products aimed to specific customers instead of mass markets (Seppänen et al., 2017). In any case, early and close customer co-operation is very beneficial for a startup figuring out its first product. Reasonable amount of effort should be put in finding potential customers willing to co-operate with a startup –willingness to work with a startup having only a vague idea of a product and a Minimum Viable Product (MVP), may be low among the potential customers.
- ***Rapid progress*** refers to a startup's need to validate the value of the product idea as rapidly as possible. Rapid progress along with earliness is a key means to avoid waste in Lean startup thinking, but the need to speed up the product development is a common trend in the software industry, as discussed above in section 5. The need for a rapid progress and close customer co-operation are the drivers behind one of the key concepts of Ries' lean startup, building the MVP.
- ***Measurement of the value*** of the business case is done by the customer on the basis of that MVP – and only in that way. An MVP should be at the same time easy and fast to create, but functional enough to be a valid basis for an evaluation by the targeted customer(s). What the MVP is at the end, is left fairly open. Ries' book addresses MVPs by introducing examples. That is understandable when aiming at covering a big variety of startup with the same principles, but in a practical situation the MVP must be well figured out to fit to the innovation, to the measurements purposes, and to the overall context.
- ***Learning*** refers to the experiences and opinions collected from the customer(s) concerning the value of the product idea. How the learning is gained, is covered in Ries' original book by defining such terms as validated learning and actionable metrics. Validated learning is a result of validated experiments with the latest version of the MVP and the real customer(s). As the experiments are the source of validated learning, a key issue, besides the definition of the MVP, is careful planning of the experiments in which the value of the MVP is measured.
- ***Corrective actions*** refer to the conclusions drawn from learning. The corrective actions are crystallized in Lean startup thinking into two clear alternatives, changing the idea or continuing its development, called in Ries' terminology pivoting or persevering. The criteria for pivoting or persevering is summarized by Ries as so-called actionable metrics as an opposite of vanity metrics. The actionable metrics should be accessible and audible. The point in putting such attributes as actionable and vanity on the metrics used for decision making is to highlight courage of a startup's key persons to explore the innovation in question in a critical and objective way. The positive feelings or bride of the innovator(s) should not lead to development of products that turn out to be waste. Measuring and learning in Lean startup thinking can be seen as a means to identify difficulties and find reasons not to persevere but pivot, as studied in (Bajwa et al., 2016).
- ***Iteration*** refers to the principle that the whole process of building MVP(s), measuring, and learning, has to be repeated until the aimed product gains the functionality and characteristics that guarantee the customer acceptance. This principle represents the old idea of continuous and incremental development, applied in Lean

startup thinking on the customer-centric validation of an innovation. The same continuous and incremental way of doing is proposed also for the growth stage of a startup, when the first product(s) has successfully been launched to the markets and the startup is seeking growth. A broader view on the practices of continuous and incremental development is presented in the earlier sections.

Ries encapsulates the above characteristics in a simple, cyclic process containing three steps, **1) Build, 2) Measure, and 3) Learn**, BML cycle (see Figure 18). After the three steps, the startup evaluates the results of the steps and makes a decision whether to keep the idea, persevere, or to modify it, pivot. In the former case the development of the idea to a product continues, while in the latter case the idea is modified and the build-measure-learn steps are repeated. Following this process should prevent a startup wasting its resources in developing products without market demand.

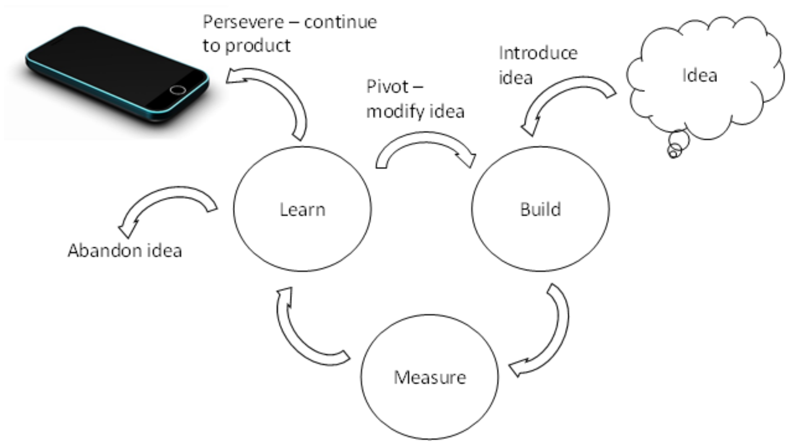


Figure 18. The BML cycle.

7.2 Derivatives and further developments

Ries' lean startup thinking was rapidly accepted not only by the practitioners but by the academia, too. A set of variants based on similar thinking were presented. Stainert and Leifer presented fairly similar ideas in their Hunter-Gatherer model (Steinert and Leifer, 2012), which was further tuned by Nguyen-Duc et al. (2015). While the Hunter-Gatherer model was at a fairly high and philosophical level, Bosch et al. (2013) and Björk et al. (2013) presented in 2013 a development model called Early Stage Software Startup Development Model (ESSSDM), claimed to be an operationalization of Lean thinking on software startups, see figure 19.

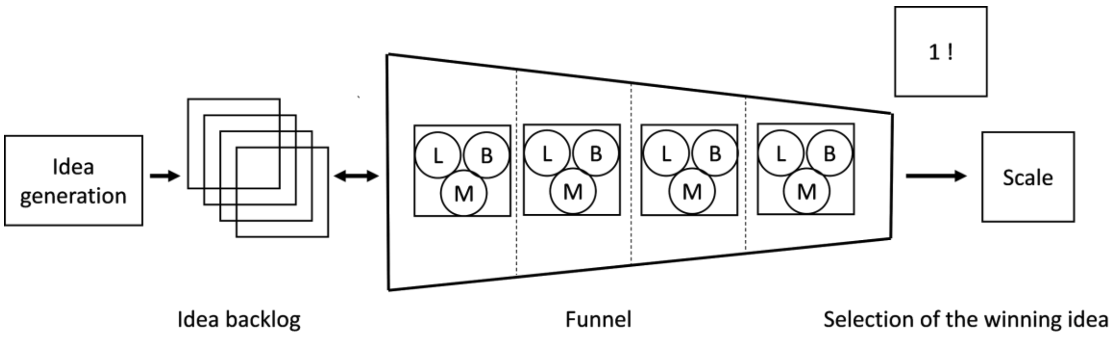


Figure 19. The ESSSDM model.

The ESSSDM model was developed to bring the ideas of Ries closer to the practical work in software startups by addressing such areas as managing multiple product ideas in a product portfolio, validation criteria of the product ideas, and techniques to use for validating them. The ESSSDM model, with its focus on product portfolios, is not principally different from the original lean startup thinking. A product portfolio including alternative product ideas, means validating several ideas in parallel or in series. However, a smaller startup may not have resources for parallel evaluation of several product ideas, as strongly recommended in the ESSSDM model. An idea portfolio may also be difficult from the focus and motivation perspectives: founding a company and taking the responsibility and risks tied to it requires courage (Seppänen et al., 2016). A strong belief on a single idea may be a better catalyzer for the needed courage than a set of ideas, which possibly are not as credible.

7.3 Lean startup and other startup research

Starting fairly early Lean startup thinking has been combined with other research paths on innovation and entrepreneurship. Müller and Thoring studied in 2012 differences and similarities of Lean startup model and design thinking (Müller and Thoring, 2012). The comparison highlighted, in an interesting way, certain characteristics of Lean startup thinking: 1) it assumes that a product vision exists e.g. a Lean startup is driven by a product vision of the startup founders, 2) it is customer-centered instead of user-centered, and 3) its focus is on the business model for a product idea – not on user-desirability or technical feasibility of the product (see figure 20).

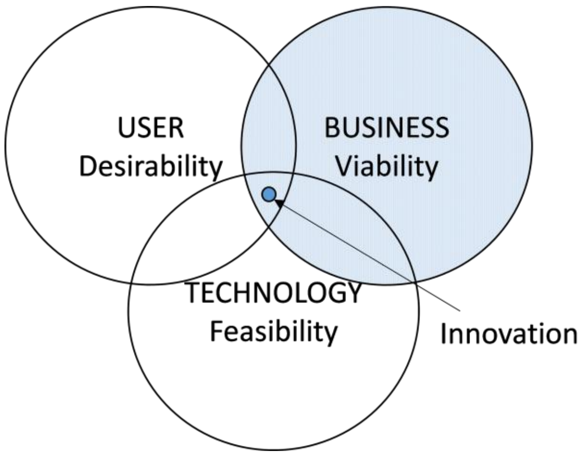


Figure 20. Criteria of successful innovation and the focus of lean startup by Müller and Thoring

The Lean startup thinking focuses on creating the business case and on validating product ideas from the business perspective. The customer acceptance of the idea is defined as the criterion of the waste. Practitioners having experience in product development know that the business feasibility is only one dimension of the whole. The technical feasibility and user desirability must be evaluated, too. Putting too much effort on an idea that is too expensive or too difficult to realize with existing technology, or difficult to use, is waste even when the idea has a real market potential (Seppänen et al., 2017). Technical feasibility and user desirability are broad areas where the weight of different criteria vary according to the context and purpose of the product: a data communications product must provide the user with speed and reliability, an interactive product with desirable user experience, understandability and easiness of use, and a banking system with data security.

An example of a broader-focus validation of the idea is the work of Hokkanen et al., combining the lean startup principles with the user experience of the targeted product (Hokkanen et al., 2016; Hokkanen et al., 2015; Hokkanen and Leppänen, 2015). Hokkanen's work complements the Lean startup from the user-centered view by highlighting the importance of a good user experience in an early stage – as a part of the MVP. According to her, an MVP must be mature enough and its user experience desirable enough to help selling the idea and collecting meaningful feedback from the customer(s). Following the original Lean startup thinking, she recommends early building of communication channels to real users, implementing usability tests before measuring, and applying planned methods to collect the feedback necessary for correct learning.

7.4 Summary of lean startup movement

As mentioned above, Lean startup thinking is vision-driven – there is an innovation, the value of which has to be measured. The startup is founded to realize the innovation to a product. Other research (Seppänen et al., 2017) has shown that many times it really is the case. A typical story of a startup is that a single person gets an idea that he/she believes in, gathers possibly a team, preferring old friends and other trusted people, and founds a company (Seppänen et al., 2016). Sometimes, founding is preceded by a long period of deliberation before the final decision for a startup is made. Of course, there are also different cases, where the need for a certain product or service is clear and the business case is easier to figure out. That does not, however, decrease the value of the basic idea of the Lean startup thinking, better to evaluate than to develop waste.

The works by Bosch et al., Björk et al., Müller and Thoring, and Hokkanen et al. are examples of how the Lean startup thinking, defined at a high abstraction level and trying to address a big variety of different startups, needs additional stuff around the principles. However, the principles, validating before progressing, iterating and improving continuously, seeking learning from real customers, being critical not vain, and avoiding waste, are good guidelines for any product development. The principles are broadly applicable also outside the original business-related focus, and their value is intuitively understandable. The basic message is: *keep on learning – do not believe that you know the value of your idea without measuring it*.

8. Miscellany

Before closing this chapter, we discuss two areas that have been the focus of our recent research in Agile and Lean software development processes, and we believe are useful for the readers. On the one hand, we focus on software metrics that are usually employed in the context of Agile and Lean software development. These metrics are important not only for understanding the situation of the product under development, but also for guiding the transformation towards Agile and Lean. On the other hand, we discuss on the topic of technical debt, which has become popular in the context of Agile software development.

8.1 Metrics in Agile and Lean Software Development

Our systematic literature review about the use of software metrics in Agile software development (Kupiainen, Mäntylä and Itkonen, 2015) revealed both similarities and differences between Agile and traditional software development. The top goals of both development approaches are similar, that is, to improve productivity and quality. The majority of the metrics are in one way or the other tied to those goals. However, we found value-based differences in means how those goals are to be reached. Traditional software development utilizes software metrics from the viewpoint of scientific management by Taylor, i.e. controlling software development from outside. The traditional measurement programs use ideas stemming from industrial product lines and manufacturing industry such as Six Sigma. Traditional metrics programs were often large-scale corporate initiatives, which emphasized following a pre-given plan. Agile principles, such as empowering the team, focusing on rapid customer value, simplicity, and willingness to embrace changes, form a contrast to the traditional measurement programs.

Figure 21 shows the top six metrics used in Agile software development. We selected the top metrics based on the number of occurrences and the perceived importance of each metric in the sources (for more details see Kupiainen, Mäntylä and Itkonen, 2015).

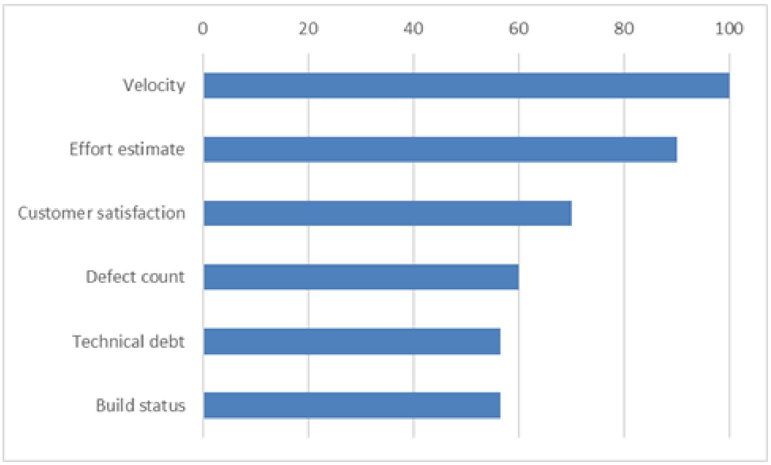


Figure 21. Top-six metrics used in Agile software development

Velocity was found as the most important software metric in Agile development. Velocity indicates how fast is the team working or how much a team can achieve in a certain time-boxed period, e.g. in a sprint of four weeks. Velocity is used in sprint planning. Velocity is also used to determine different scopes for the sprint such as having a minimum scope of features that can be delivered for sure and having a stretch goal for the iteration. Having two different scopes can help in customer communication as the minimum can be promised and the maximum indicates all the changes that may happen during a sprint. Velocity enables finding bottlenecks in a software process where progress is slower than desired. Making process improvement or automation to these parts results in an excellent cost benefit ratio for the improvement. However, our study also found cases where attempts to maintain velocity led to cutting corners and lowered the product quality.

Effort estimate, which indicates the anticipated work cost of a particular feature or story, was found as the second most important metric. Effort estimates are important in feature prioritizations as all features with equal value can be sorted by the effort estimates to provide a sprint scope with the highest possible value. However, *No Estimates* movement started by Ron Jefferies, approximately says that making effort estimates can be counterproductive as the accuracy of the estimates is too poor to make the effort spent in estimation worthwhile. Nevertheless, effort estimates were often used and found important in the sources of our literature review.

Customer satisfaction was found as the 3rd ranked metric although it could be claimed that it should be the most important due to the customer centric nature of Agile software development. Yet, it was less frequently mentioned in our sources than Velocity or Effort Estimate. Customer satisfaction is a metric with many possible operationalizations. We found several measures of customer satisfaction such as the number of change requests by the customer, net promoter score (the probability that a customer would recommend the product or producer to another potential customers), Kano analysis (quality model that distinguishes between must-be, one-dimensional, attractive, indifference and negative quality), and post release defects found by the customers. We found that most successful projects measured customer satisfaction more often than other projects.

We find that the core books giving advice on Agile software development are missing some core software engineering metrics such as the *defect count*. In our review of industrial empirical studies, it was ranked in the fourth place indicating its importance in Agile software development. Defect count can have many uses. It may be a measure of software testing as the number of defects found per time unit in testing measures in-house software quality. Furthermore, the defects experienced in customer usage can be a measure of the software quality in the wild.

Technical debt, which is covered in more detail next, was among the top metrics used by Agile teams. Technical debt hinderers day-to-day software development and reduces velocity. In one company, a technical debt board was used to make technical debt visible in order to ensure that enough resources and motivation were given to removing the debt. The board was also used to make plans how to remove technical debt. The board made sure that developers picked the highest priority technical debt items instead of cherry picking lower priority but more interesting tasks.

Build status is an important measure in Agile software development. It is an indication of the agile principle working code. If builds are not passing, this indicates that no progress according to agile principles has been made. Build status often encompasses quality gates in addition to the simple assertion that the software has been successfully built. It can include the results of automated unit testing. Frequently, successful build requires that certain end-to-end automated acceptance test are passed. When working on software with very high reliability requirements, even violations from static code analysis tools such as, Findbugs or Coverity, may be treated as build breakers.

We also studied the purposes of using metrics in Agile software development, which are depicted in Figure 22.

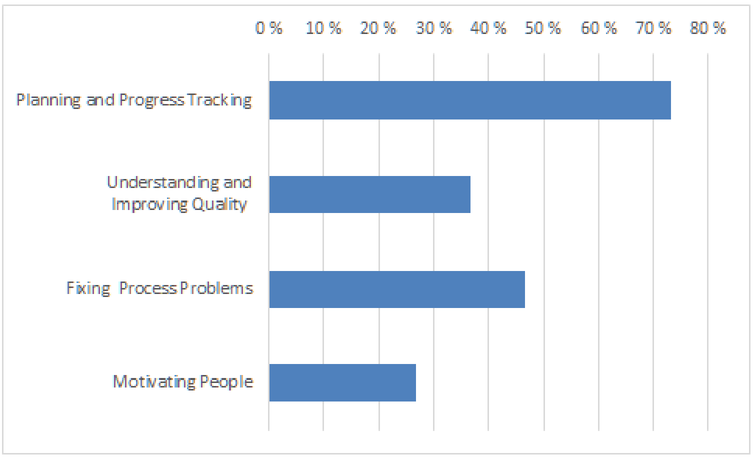


Figure 22. Reasons for using metrics in Agile teams.

We found that over 70% sources used metrics for planning and progress tracking of sprints or projects. Understanding and improving quality was found as a reason for using metrics in roughly one third of the studies. Fixing process problems inspired roughly half of the studies in using metrics. Finally, motivating people was found as the reason for using metrics in 25% of the studies, for example displaying a technical debt board can motivate people to reduce technical debt. The percentage does not add up to 100% because many studies had multiple motivations in using metrics.

8.2 Technical Debt in Agile and Lean Software Development

The Agile Manifesto highlights '*continuous attention to technical excellence and good design enhances agility*'. Similarly, XP has '*coding standards*' as one of its key practices. Still, the scientific literature highlights that Agile software development appears to be more prone to technical debt compared to traditional software development (Guo et al., 2014). This section develops into the concept of technical debt and describes different strategies that can be applied to manage technical debt in Agile software development.

8.2.1 What is technical debt?

Technical debt is a metaphor used in software development to communicate the consequences of poor software development practices to non-technical stakeholders. Ward Cunningham firstly introduced it in 1992 (Cunningham, 1992). The concept works like a financial debt that needs to be paid back with some extra interest but applied to the software development process. Technical debt happens, for example, when developers focus only on delivering new functionality at the end of each sprint and lack time in refactoring, documentation, and other aspects of the project infrastructure. If these aspects are not considered in the sprint backlog and ignored, the software will end up growing less understandable, more complex, and harder to modify. We can view this system degradation as a debt that developers owe to the system (Shull, 2011). Cunningham (1992) explained that a little debt can speed up software development in the short run. However, this benefit is achieved at the cost of extra work in the future, as if paying interest on the debt. The debt increases when it is not promptly repaid. Time spent on *not-quite-right* code counts as interest on that debt (Cunningham, 1992).

Technical debt has been usually associated to poorly written code. For example, code that is unnecessarily complex, poorly organized or includes unnecessary duplications will require future work (e.g. refactoring), which is a form of technical debt. However, as time passes by, technical debt has evolved to embrace different aspects of software development that range from architectural deterioration and design problems to testing, documentation, people, and deployment issues (Tom et al., 2013). Another example of technical debt in the area of testing is when tests scripts are not properly maintained, leading to additional manual testing. In general, technical debt can be manifested in any aspect of the software development process that relates to internal system qualities, primarily maintainability and evolvability (Kruchten, 2016).

8.2.2 Technical debt: debt or investment?

Technical debt seems to be, therefore, something negative, which often results in undesirable consequences such as system degradation, increased maintenance costs or decreased productivity. However, technical debt is not only about technical shortcuts because pressures of the moment or accumulation of unfinished work. It can be used also constructively as a framework to reflect about different alternatives and make informed decisions about what to pass up (Shull, 2011). As everything in life, there are always trade-offs in software development, which, unfortunately, is not a world of infinite resources. Software companies need to decide where to invest their finite resources to get the maximum value for their business and their customers. Not doing something good today will allow doing something else that is also good and, at the end, will pay back better in the future (Shull, 2011). For example, we may decide to sacrifice certain quality level in the next release because we prefer to include a feature that will provide a competitive advantage. Getting to the market to catch an important business opportunity may influence the decision to take on technical debt. Overall, technical debt is seen as a balance between software quality and business reality (Lim et al., 2012).

As represented in Figure 23, technical debt can be classified in two groups: reckless and prudent debt (Fowler, 2009). Reckless debt is the *'bad'* or unhealthy debt; the debt that is considered negative. It is incurred as the result of a shortage, sloppy programming or poor development. It can be inadvertent, for example, because of poor technical knowledge, or deliberated, if the debt is a result of sloppy or careless programing. However, there is also a healthy or *'good'* debt, the prudent debt. Prudent technical debt is proactively incurred in order to get some benefit. For example, achieving a quicker release. In this situation, the team deliberately decides to take on technical debt and has plans to deal with consequences. Prudent debt is, therefore, related to intentional decisions to trade off competing investments during development. Some authors argue that prudent debt could be also inadvertent (Fowler, 2009), in the sense that software development is usually a learning activity. Software engineers may learn different alternatives during the development process that would have been better options from a technical point of view.

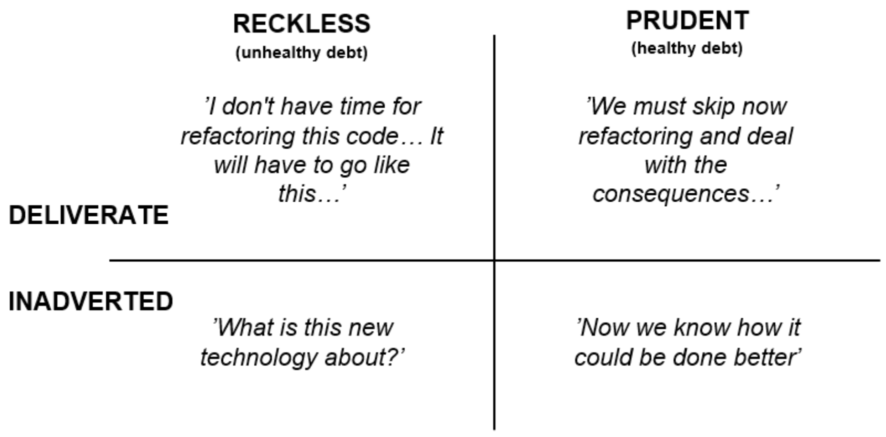

RECKLESS
(unhealthy debt)
PRUDENT
(healthy debt)
DELIVERATE
'I don't have time for refactoring this code… It will have to go like this…'
'We must skip now refactoring and deal with the consequences…'
INADVERTED
'What is this new technology about?'
'Now we know how it could be done better'

Figure 23. Reckless vs. prudent debt - based on Fowler's quadrant of technical debt (Fowler, 2009).

Overall, technical debt effects are not always negative. The problem with technical debt comes when it is not properly managed or gets uncontrolled. The key is to make informed decisions and be prepared for the consequences. When properly managed and controlled, technical debt can be utilized as an Agile strategy to gain business opportunities and be paid back later. Next section describes different strategies to manage technical debt in an Agile context.

8.2.3 Strategies to manage technical debt in agile software development

Besides the emphasis on quick delivery, which is frequently identified as one of the most important causes for accumulating technical debt in Agile (Behutiye et al., 2017), other aspects can also generate technical debt in Agile. For example: 1. architecture and design issues including inflexibility in architecture, poor design, and suboptimal up-front architecture and design solutions, 2. inadequate test coverage and issues with test automation, 3. lack of understanding of the system being built, 4. oversight in estimations of sprints and, in general, unrealistic estimations, and 5. inadequate refactoring (Behutiye et al., 2017). The consequences are quite straightforward, system quality degradation, which increases maintenance costs and reduces productivity, market loss/hurt business relationships and even complete redesign or rework of a system (Behutiye et al., 2017). Therefore, it is important to learn strategies that help manage technical debt in Agile and use it in a healthy way. Our systematic literature review on technical debt in Agile software development (Behutiye et al., 2017) found the following solutions to manage technical debt and balance rapid delivery and quality development:

- *Refactoring* is the most common technique applied to reduce technical debt in Agile software development. Refactoring consists in restructuring the code base or system architecture without altering the external behavior of the system. The key idea is to redistribute classes, variables and methods across the class hierarchy to simplify the underlying logic of the software and eliminate unnecessary complexity. Refactoring should be a continuous activity in Agile and rapid software development. Nowadays, many development environments include automated support for refactoring, at least for mechanical aspects of refactoring. A comprehensible overview of software refactoring, including an illustrative example on how it is conducted in practice, can be found in (Mens and Tourwé, 2004).
- *Measuring/visualizing technical debt and communicating its consequences*. Measuring technical debt is fundamental to make it visible. Non-visible technical debt goes usually unnoticed, being a clear risk for the

development (unhealthy debt). As presented earlier, technical debt is among the top metrics used by Agile teams. Unmeasured technical debt makes it hard for development teams to make a strong case for business stakeholders to invest in technical debt fixes. Overall, technical debt should be transparent. Design decisions related to technical debt and architecture dependences should be clear for everyone. The literature suggests different strategies to monitor technical debt and make it visible, such as: keeping track of a list of architectural and design decisions in a backlog (Bellomo et al., 2013; Abrahamsson et al., 2010), using technical debt visualization boards (Nord et al., 2012a; Nord et al., 2012b; Letouzey, 2012; dos Santos et al., 2013), visualizing technical debt using the 'code Christmas tree' (Kaiser and Royse, 2011), pie and bar charts to visualize and manage technical debt (Power 2013), and technical visualization tools like Ndpend to detect code violations (Hanssen et al., 2010).

- *Planning for technical debt and prioritizing technical debt when needed*. Using technical debt proactively means that we know that we are taking technical debt and we plan how to deal with the consequences generated by our decision. Advance planning for technical debt involves assigning preemptive efforts to address it when needed by allocating resources in advance for tasks such as code cleanup, removal of design shortcuts and refactoring. Dedicated teams, which are particularly responsible for technical debt reduction, can also be allocated when needed. Measuring is essential here in order to objectively determinate 'when needed'. Optimistic estimations by Agile teams lead to technical debt. Technical debt items should be explicitly included in the product backlog so resources are explicitly allocated for these tasks. Moreover, increasing team's' estimation ability leads to reduce reckless technical debt.
- The *Definition of Done* (DoD) supports also controlling technical debt in Agile. Employing a common DoD in different levels, such as story, sprint, and release, helps achieve a common understanding on technical debt and manage it strategically. For example, definition of the *right-code* can be include in the DoD. The DoD can be also used during sprint planning meetings and sprint reviews to reveal technical debt issues and plan how to deal with consequences. Overall, the DoD helps with establishing an acceptable level of technical debt.
- *Automation*, and particularly, test automation helps also reduce technical debt and increase test coverage.

A more detailed analysis of the scientific literature on technical debt in Agile software development is reported in our systematic literature review on the topic (Behutiye et al. 2017).

9. Conclusions and Future Directions

Agile software development is well-established in software industry. In this book chapter, we have gone through the main elements of Agile and Lean software development processes, from the very fundamental principles and values that guide these approaches to the concrete methods and practices existing in the literature to implement the fundamentals in practice. Particularly, we have discussed the main characteristics of 'agility' and 'leanness' as they emerged in the manufacturing domain (Section 3). The five original principles of Lean thinking (Womack and Jones, 1996) are: 1) *value*, seem from a customer's perspective; 2) *value stream*, as the stream of production activities in which every activity delivers customer value; 3) *flow*, which provides continuity to the value stream; 4) *pull*, to make sure that production is guided only by demand; and 5) *perfection*, the pillar for continuous improvement. Agility, on the other hand, is characterized by four capabilities (Sharifi and Zhang 1999): 1) *responsiveness*, to identify and respond fast to changes, 2) *competence*, to have all capabilities to achieve productivity, efficiency and effectiveness, 3) *flexibility*, to make changes in products and achieve different objectives using the same facilities, and 4) *speed*, to reduce time-to-market.

The closeness of ideas between both paradigms have favoured their combination (known as le-agility). Although this combination was taken with caution in manufacturing - as it is described in Section 3 - the journey of Agile and Lean in the software domain has followed a quite different path; a path characterized by a symbiosis between Agile and Lean in which limits are not clearly established. The foundations of Agile software development processes were established in the Agile Manifesto through four values and twelve principles. These foundations created the basis for Agile software development methods that implement those values and principles in practice (e.g. Scrum and XP). Agile software development emerged as the best possible solution to face the challenges posed by a turbulent and dynamic software market. However, although Agile processes provided important benefits at development team level, they were not enough to operate a whole organization. Thus, Lean software development emerged as a way of scaling Agile and, in general, complementing Agile software development processes. More recently - as discussed in Section 5 - Agile and Lean software development processes have been extended to continuous delivery and deployment. The idea is that software development becomes a continuous process that is able to dynamically adapt to business changes and discover new business opportunities.

If we had to highlight two essential characteristics of current software processes, those would be *speed* and *learning*. Speed is essential because it does not only allow obtaining a competitive advantage, but it also favours short feedback loops. Short feedback loops are the basis for the second essential characteristic of current software processes, learning, and more concretely, *validated learning*. Learning is important because it is the key element that allows software companies to adapt to business changes. The goal is to learn (instead of predicting) where value is through practices such as continuous experimentation, which guide product development and accelerate innovation. DevOps (discussed in Section 6), and Lean start-up, (discussed in Section 7), are key elements to achieve speed and validated learning.

How do we see the future of software development processes and, particularly, Agile software development? As the software development industry evolves, versatile and flexible software development processes are becoming essential to cope with current market and customers' demands. Old-fashioned software development processes, such as the waterfall model, worked well under the software industry conditions where they were developed. However, they are too complex to surface innovation in current software markets, as they are not flexible and fast enough for many software companies. Thus, Agile is turning into a mainstream methodology for software development. Although, initially it was taken with care by many software sectors, in which Agile was seen just as the latest fad that, as any fad, would have a short life, it is quite clear now that Agile won the recognition battle and it is here to stay. Nowadays, Agile is not only used by pioneers in innovative projects, but it has spread across a broad range of industries. This is particularly true in web, mobile applications or services domains, where applications are frequently developed in weeks or months, rather than years.

Years ago, people were not as exposed to software products as they are now. Nowadays, we are using software every single day of our lives. In our houses, our jobs, hospitals, schools... Smartphones, watches, fridges and cars are obvious devices containing software. However, smart microprocessors are embedded in almost every device that you can imagine: thermostats, garage doors, front doorbells... Did you know that smart microprocessors are starting to be installed in walls and foundations of houses to monitor stress fractures, watering and weakening in the structure of buildings? It is clear that the software development industry is growing fast. Many innovations are emerging and will continue to emerge. For example, we think that trends like the internet of things (IoT) will shape the software development processes of more traditional industries in the near future. The IoT is much more than hardware and connectivity. Software is needed to run and connect all these smart devices collecting and exchanging data, and connecting every aspect of our lives. Moreover, with IoT there are increasing demands in reliability and in particular to security. When software controls physical world products, security holes must be quickly and continuously patched. Agile, Lean and rapid software development will play a key role in such development.

Both the software development industry and the research community are being very active in shaping and extending Agile and Lean to processes such as rapid software development. For example, continuous processes are under constant exploration to identify ways to safely get speed in software development. Ways to activate a continuous - real time, learning capability to allow companies to adapt to business dynamics, such as continuous experimentation, are also being increasingly explored. Software processes must be value-driven. Better understanding on the concept of value will provide the means to improve value-based decision making as well.

References

Abrahamsson P, Salo O, Ronkainen J & Warsta J (2002) Agile Software Development Methods: Review and Analysis. VTT Publications 478.

Abrahamsson P, Conboy K & Wang X (2009) 'Lots done, more to do': the current state of agile systems development research. European Journal of Information Systems 18(4): 281–28.

Abrahamsson, P., Babar, M. A., & Kruchten, P. (2010). Agility and architecture: Can they coexist? IEEE Software, 27(2), 16–22.

Agresti WW (1986) New paradigms for software development: tutorial. IEEE Computer Society Press.

Ahmad M O, Markkula J & Oivo, M (2013) Kanban in software development: A systematic literature review. Proceedings of the 39th Euromicro Conference series on Software Engineering and Advanced Applications (SEAA) Santander, Spain.

Anderson DJ (2010) Kanban, Blue Hole Press.

Antinyan, V., Staron, M., Meding, W., Osterstrom, P., Wikstrom, E., Wranker, J., Henriksson, A., Hansson, J. (2014). Identifying risky areas of software code in agile/lean software development: an industrial experience report. In: 2014 Soft- ware Evolution Week-IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE). IEEE, pp. 154–163

Arisholm, E., Gallis, H., Dyba, T., & Sjoberg, D. I. (2007). Evaluating pair programming with respect to system complexity and programmer expertise. IEEE Transactions on Software Engineering, 33(2).

Agarwal, P., 2011. Continuous Scrum: agile management of SaaS products. In: Pro- ceedings of the 4th India Software Engineering Conference. ACM, pp. 51–60.

Babb, J., Hoda, R., & Norbjerg, J. (2014). Embedding reflection and learning into agile software development. IEEE software, 31(4), 51-57.

Bajwa, S. S., Wang, X., Duc, A. N., & Abrahamsson, P. (2016). “Failures” to be celebrated: an analysis of major pivots of software startups. Empirical Software Engineering, 1-36.

Balalaie, A., Heydarnoori, A., & Jamshidi, P. (2016). Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture. *IEEE Software*, *33*(3), 42–52

Basili VR & Turner AJ (1975) Iterative enhancement: A practical technique for software development. Software Engineering, IEEE Transactions on (4): 390–396.

Bass, L., Weber, I., & Zhu, L. (2015). *DevOps: A Software Architect’s Perspective*. Addison-Wesley Professional.

Beck K, Beedle M, van Bennekum A, Cockburn A, Cunningha, W, Fowler M, Grenning J, Highsmith J, Hunt A, Jeffries R, Kern J, Marick B, Martin RC, Mellor S, Schwaber K, Sutherland J & Thomas D. (2001) Manifesto for Agile Software Development, http://www.agilemanifesto.org/.

Beck K & Andres C (2004) Extreme programming explained: embrace change, Addison-Wesley Professional.

Beck, K. (2003). *Test-driven development: by example*. Addison-Wesley Professional.

Beck K & Boehm B (2003) Agility through discipline: A debate. Computer 36(6): 44–46.

Behutiye, W. N., Rodríguez, P., Oivo, M., & Tosun, A. (2017). Analyzing the concept of technical debt in the context of agile software development: A systematic literature review. Information and Software Technology, 82, 139-158.

Bellomo, S., Nord, R.L., Ozkaya, I. (2013a). Elaboration on an integrated architecture and requirement practice: prototyping with quality attribute focus. In: 2013 2nd International Workshop on the Twin Peaks of Requirements and Architecture (TwinPeaks). IEEE, pp. 8–13.

Bellomo, S., Nord, R. L., & Ozkaya, I. (2013b). A study of enabling factors for rapid fielding combined practices to balance speed and stability. pp. 982–991.

Benefield, R., 2009. Agile deployment: lean service management and deployment strategies for the SaaS enterprise. In: 42nd Hawaii International Conference on System Sciences, 2009, HICSS’09. IEEE, pp. 1–5

Biffl, S., Aurum, A., Boehm, B., Erdogmus, H., & Grünbacher, P. (Eds.). (2005). Value-based software engineering. Springer Science & Business Media.

Björk, J., Ljungblad, J., & Bosch, J. (2013). Lean Product Development in Early Stage Startups. In IW-LCSP@ ICSOB (pp. 19-32).

Boehm B (1988) A spiral model of software development and enhancement. Computer 21(5): 61–72.

Boehm B (2002) Get ready for agile methods, with care. Computer 35(1): 64–69.

Bosch, J. (2017). Speed, Data, and Ecosystems: Excelling in a Software-Driven World. CRC Press.

Bosch, J. (2012). Building products as innovation experiment systems. In: Software Business. Springer, pp. 27–39.

Bosch, J., Olsson, H. H., Björk, J., & Ljungblad, J. (2013). The early stage software startup development model: a framework for operationalizing lean principles in software startups. In Lean Enterprise Software and Systems (pp. 1-15). Springer, Berlin, Heidelberg.

Bremner B, Dawson C, Kerwin K, Palmeri C & Magnusson P (2003) Can anything stop Toyota? Bus Week 117.

Brown, A. W., Ambler, S., & Royce, W. (2013). Agility at scale: economic governance, measured improvement, and disciplined delivery. In *Proceedings of the 2013 International Conference on Software Engineering* (pp. 873-881). IEEE Press.

Callanan, M., & Spillane, A. (2016). DevOps: Making It Easy to Do the Right Thing. *IEEE Software*, *33*(3), 53–59.

Castells M (2011). The rise of the network society: The information age: Economy, society, and culture, Wiley-Blackwell.

Chen, I. M. (2001). Rapid response manufacturing through a rapidly reconfigurable robotic workcell. Robotics and Computer-Integrated Manufacturing, 17(3), 199-213.

Chen, L. (2017). Continuous Delivery: Overcoming Adoption Challenges. *Journal of Systems and Software*. http://doi.org/10.1016/j.jss.2017.02.013

Christopher, M. & Towill, D. R. (2000) Supply chain migration from lean and functional to agile and customised. Supply Chain Management: An Int. Journal 5(4): 206–213.

Cito, J., Leitner, P., Fritz, T., & Gall, H. C. (2015). The making of cloud applications: an empirical study on software development for the cloud. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015* (pp. 393–403)

Cito, J., Oliveira, F., Leitner, P., Nagpurkar, P., & Gall, H. C. (2017, May). Context-based analytics: establishing explicit links between runtime traces and source code. In Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track (pp. 193-202). IEEE Press.

Claps, G. G., Berntsson Svensson, R., & Aurum, A. (2015). On the Journey to Continuous Deployment: Technical and Social Challenges Along the Way. *Information and Software Technology*, *57*, 21–31

CMMI (2010) Capability maturity model integration 1.3, Carnegie Mellon Software Engineering Institute, http://www.sei.cmu.edu/cmmi/.

Cobb, C. G. (2011). Making sense of agile project management: balancing control and agility, Wiley.

Cockburn A & Highsmith J (2001) Agile software development, the people factor. Computer 34(11): 131–133.

Conboy K, Coyle S, Wang X & Pikkarainen M (2011) People over Process: Key Challenges in Agile Development. IEEE Software 28(4): 48–57.

Conboy, K (2009) Agility from first principles: Reconstructing the concept of agility in information systems development. Information Systems Research 20(3): 329–354.

Cockburn, A. (2004). *Crystal clear: a human-powered methodology for small teams*. Pearson Education.

Cunningham W (1992) The WyCash portfolio management system, SIGPLAN OOPS Mess. 4 (1992) 29–30, doi: 10.1145/157710.157715

Cusumano, MA (2011) Reflections on the Toyota debacle. Commun ACM 54(1): 33–35.

de Souza, LB (2009) Trends and approaches in lean healthcare. Leadership in Health Services9 22(2): 121–139.

Debois, P. (2008). Agile Infrastructure and Operations: How Infra-gile are You? In *Agile 2008 Conference* (pp. 202–207). IEEE. http://doi.org/10.1109/Agile.2008.42

Delen, D., & Demirkan, H. (2013). Data, information and analytics as services. Decision Support Systems 55(1).359-363

Donaldson L (2001) *The contingency theory of organizations*. Sage

dos Santos, P. S. M., Varella, A., Dantas, C. R., & Borges, D. B. (2013, June). Visualizing and managing technical debt in agile development: An experience report. In *International Conference on Agile Software Development* (pp. 121-134). Springer, Berlin, Heidelberg.

Dybå T & Dingsøyr T (2008) Empirical studies of agile software development: A systematic review. Information and software technology 50(9): 833–859.

Dybå T & Sharp H (2012) What's the Evidence for Lean? IEEE Software 29(5): 19–21.

Eklund, U., Bosch, J., 2012. Architecture for large-scale innovation experiment systems. In: 2012 Joint Working IEEE/IFIP Conference on Software Architecture (WICSA) and European Conference on Software Architecture (ECSA). IEEE, pp. 244–248.

Eisenhardt KM & Martin JA (2000) Dynamic capabilities: what are they? Strategic Manage J 21(10–11): 1105–1121.

Elbanna, A., & Sarker, S. (2016). The Risks of Agile Software Development: Learning from Adopters. *IEEE Software*, *33*(5), 72–79

Elberzhager, F., Arif, T., Naab, M., Süß, I., & Koban, S. (2017). From Agile Development to DevOps: Going Towards Faster Releases at High Quality – Experiences from an Industrial Context (pp. 33–44). Springer.

Fagerholm, F., Guinea, A.S., Mäenpää, H., Münch, J., (2014). Building blocks for continuous experimentation. In: Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering (RCoSE 2014), Hyderabad, India.

Feitelson, D.G., Frachtenberg, E., Beck, K.L., (2013). Development and deployment at Facebook. IEEE Internet Comput. 17 (4), 8–17.

Fitzgerald, B., & Stol, K. J. (2014). Continuous software engineering and beyond: trends and challenges. In Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering (pp. 1-9). ACM.

Fitzgerald, B., & Stol, K. J. (2017). Continuous software engineering: A roadmap and agenda. *Journal of Systems and Software*, *123,* 176-189.

Fowler, M. (2009). Technical debt quadrant, http://martinfowler.com/bliki/TechnicalDebtQuadrant.html, accessed: 29.06.2017.

Franch, X., Ayala, C., López, L., Martínez-Fernández, S., Rodríguez, P., Gómez, C., Jedlitschka, A., Oivo, M., Partanen, J., Räty, T., & Rytivaara, V. (2017). Data-driven Requirements Engineering in Agile Projects: The Q-Rapids Approach. Proceedings of the 2nd International Workshop on Just-In-Time Requirements Engineering: Dealing with Non-Functional Requirements in Agile Software Development, Lisbon, Portugal, 2017, pp. 1-4.

Freeman P. (1992). Lean concepts in software engineering. IPSS-Europe International Conference on Lean Software Development, Stuttgart, Germany. : 1–8.

Fujimoto T. (1999). Evolution of Manufacturing Systems at Toyota. Oxford University Press, Inc., New York (1999).

Garousi, V., & Mäntylä, M. V. (2016). Citations, research topics and active countries in software engineering: A bibliometrics study. Computer Science Review, 19, 56-77.

Goel, A., Chopra, B., Gerea, C., Mátáni, D., Metzler, J., Ul Haq, F., Wiener, J. (2014). Fast database restarts at Facebook. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data. ACM, pp. 541–549

Gotel, O., & Leip, D. (2007). Agile Software Development Meets Corporate Deployment Procedures: Stretching the Agile Envelope. In *Agile Processes in Software Engineering and Extreme Programming* (pp. 24–27). Berlin, Heidelberg: Springer

Guo, Y., Spínola, R. O., & Seaman, C. (2016). Exploring the costs of technical debt management---a case study. *Empirical Software Engineering*, *21*(1), 159-182.

Guzmán, L., Oriol, M., Rodríguez, P., Franch, X., Jedlitschka, A., & Oivo, M. (2017). How Can Quality Awareness Support Rapid Software Development?–A Research Preview. In International Working Conference on Requirements Engineering: Foundation for Software Quality (pp. 167-173). Springer.

Hanssen, G., Yamashita, A. F., Conradi, R., & Moonen, L. (2010). Software entropy in agile product evolution. In *System Sciences (HICSS), 2010 43rd Hawaii International Conference on* (pp. 1-10). IEEE.

Heinrich, R., van Hoorn, A., Knoche, H., Li, F., Lwakatare, L. E., Pahl, C., & Wettinger, J. (2017, April). Performance Engineering for Microservices: Research Challenges and Directions. In Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion (pp. 223-226). ACM.

Highsmith, J., & Cockburn, A. (2001). Agile software development: The business of innovation. *Computer*, *34*(9), 120-127.

Highsmith, JA (2002). Agile software development ecosystems, Addison-Wesley Professional.

Hoda, R., Noble, J., & Marshall, S. (2011). The impact of inadequate customer collaboration on self-organizing Agile teams. Information and Software Technology, 53(5), 521-534.

Hokkanen, L., Kuusinen, K., & Väänänen, K. (2016). Minimum viable user experience: A framework for supporting product design in startups. In International Conference on Agile Software Development (pp. 66-78). Springer, Cham.

Hokkanen, L., Kuusinen, K., & Väänänen, K. (2015). Early product design in startups: towards a UX strategy. In International Conference on Product-Focused Software Process Improvement (pp. 217-224). Springer International Publishing.

Hokkanen, L., & Leppänen, M. (2015). Three patterns for user involvement in startups. In Proceedings of the 20th European Conference on Pattern Languages of Programs (p. 51). ACM.

Hossain, E., Babar, M. A., & Paik, H. Y. (2009, July). Using scrum in global software development: a systematic literature review. In Fourth IEEE International Conference on Global Software Engineering (ICGSE). (pp. 175-184). IEEE.

Humble, J., Read, C., & North, D. (2006). The deployment production line. In: Agile Conference, 2006. IEEE, p. 6.

Humble, J., Farley, D., 2010. Continuous Delivery: Reliable Software Releases through Build, Test, and deployment automation. Pearson Education.

Humble, J., & Molesky, J. (2011). Why enterprises must adopt DevOps to enable continuous delivery. *Cutter IT Journal*, *24*(8), 6–12

Hummer, W., Rosenberg, F., Oliveira, F., & Eilam, T. (2013). Testing idempotence for infrastructure as code. *Middleware 2013*.

Iden, J., Tessem, B., & Päivärinta, T. (2011). Problems in the interplay of development and IT operations in system development projects: A Delphi study of Norwegian IT experts. *Information and Software Technology*, *53*(4), 394–406

ISO/IEC 122007:2008(E) (2008) Systems and software engineering – Systems life cycle processes, second edition, ISO/IEC, Geneva, Switzerland.

ISO/IEC 90003:2004(E) (2004) Software engineering - Guidelines for the application of ISO 9001:2000 to computer software, ISO/IEC 90093:2004(E).

ISO/IEC 15504-1:1998 (1998) Information technology - Process assessment – Part1: Concepts and introductory guide, WG10N222.

Jacobson I, Booch G & Rumbaugh JE (1999) The unified software development process-the complete guide to the unified process from the original designers, Addison-Wesley

Järvinen, J., Huomo, T., Mikkonen, T., Tyrväinen, P., 2014. From agile software development to mercury business. In: Software Business. Towards Continuous Value Delivery. Springer, pp. 58–71.

Jones, S., Noppen, J., & Lettice, F. (2016). Management challenges for DevOps adoption within UK SMEs. In *Proceedings of the 2nd International Workshop on Quality-Aware DevOps - QUDOS 2016* (pp. 7–11).

Kaiser, M., & Royse, G. (2011). Selling the Investment to Pay Down Technical Debt: The Code Christmas Tree. In *Agile Conference (AGILE), 2011* (pp. 175-180). IEEE.

Kruchten P (2011) A plea for lean software process models. Proceedings of the 2011 International Conference on Software and Systems Process. ACM: 235–236.

Kruchten P (2016) Refining the definition of technical debt. https: //philippe.kruchten.com/2016/04/22/refining- the- definition- of- technical- debt/ (accessed 29.06.2017).

Kupiainen, E., Mäntylä M. V., Itkonen J., "Using Metrics in Agile and Lean Software Development - A Systematic Literature Review of Industrial Studies", Information and Software Technology, vol 62, June, 2015, pp. 143–163.

Kuvaja P & Bicego A (1994) BOOTSTRAP—a European assessment methodology. Software Quality Journal 3(3): 117–127.

Laanti M (2012) Agile methods in large-scale software development organisations. Applicability and model for adoption. Doctoral thesis. University of Oulu.

Ladas, C., 2009. Scrumban – Essays on Kanban Systems for Lean Software Development. Modus Cooperandi Press.

Larman, C. & Vodde, B. (2009). Scaling lean & agile development: Thinking and organisational tools for large-scale Scrum, Addison-Wesley Professional.

Larman C., & Vodde, B. (2010). Practices for scaling Lean and Agile development: large, multisite and offshore product development with large-scale Scrum. Addison-Wesley Professional.

Laukkarinen, T., Kuusinen, K., & Mikkonen, T. (2017). DevOps in regulated software development: case medical devices. *Proceedings of the 39th International Conference on Software Engineering: New Ideas and Emerging Results Track*, 15–18

Leffingwell, D. (2010). *Agile software requirements: lean requirements practices for teams, programs, and the enterprise*. Addison-Wesley Professional.

Leppänen M., Mäkinen S. V., Pagels M. E, Eloranta V. P., Itkonen J., Mäntylä M. V., Männistö T. (2015) "The Highways and Country Roads to Continuous Deployment", IEEE Software, vol. 32., no 2, pp. 64-71.

Letouzey, J. L. (2012). The SQALE method for evaluating technical debt. In *Managing Technical Debt (MTD), 2012 Third International Workshop on* (pp. 31-36). IEEE.

Liker JK (2004) The toyota way, Esensi.

Lim, E., Taksande, N., & Seaman, C. (2012). A balancing act: what software practitioners have to say about technical debt. *IEEE software*, *29*(6), 22-27.

Lwakatare, L. E., Kuvaja, P., & Oivo, M. (2015). Dimensions of DevOps. In C. Lassenius, T. Dingsøyr, & M. Paasivaara (Eds.), 16th International Conference on Agile Software Development (XP) (Vol. 212, pp. 212–217). Cham: Springer International Publishing

Lwakatare, L. E., Karvonen, T., Sauvola, T., Kuvaja, P., Bosch, J., Olsson, H. H., & Oivo, M. (2016). Towards DevOps in the embedded systems domain: Why is it so hard? In 49th Hawaii International Conference on Systems Science pp. 5437-5446. IEEE

MacCormack, A. (2001). How internet companies build software. MIT Sloan Manage. Rev. 42 (2), 75–84

Maglyas, A., Nikula, U. & Smolander, K. (2012). Lean solutions to software product management problems. Software, IEEE 29(5): 40–46.

Mandic V, Oivo M, Rodriguez P, Kuvaja P, Kaikkonen H & Turhan B (2010) What is flowing in Lean Software Development? In Proceedings of the 1st International Conference on Lean Enterprise Software and Systems (LESS), 72–84.

Mäntylä, M.V., Adams, B., Khomh, F., Engström, E., Petersen, K.: (2015) On Rapid Releases and Software Testing: A Case Study and a Semi-systematic Literature Review. Empirical Software Engineering, 25(2), 1384-1425.

Maples C (2009) Enterprise agile transformation: the two-year wall. Agile Conference, 2009. AGILE'09. IEEE: 90–95.

Marschall, M. (2007). Transforming a six month release cycle to continuous flow. In: Agile Conference (AGILE), 2007. IEEE, pp. 395–400.

Marchwinski C & Shook J (2008) Lean lexicon: a graphical glossary for lean thinkers, fourth edition, Lean Enterprise Institute.

Mason-Jones, R., Naylor, J.B., Towill, D.: Engineering the Leagile Supply Chain. International Journal of Agile Management Systems 2(1), 54–61 (2000)

Mehta M, Anderson D & Raffo D (2008) Providing value to customers in software development through lean principles. Software Process: Improvement and Practice 13(1): 101–109.

Mendes, E., Rodriguez, P., Freitas, V., Baker, S., & Atoui, M. A. (2017). Towards improving decision making and estimating the value of decisions in value-based software engineering: the VALUE framework. *Software Quality Journal*, 1-50.

Mens, T., & Tourwé, T. (2004). A survey of software refactoring. *IEEE Transactions on software engineering*, *30*(2), 126-139.

Middleton P (2001) Lean software development: two case studies. Software Quality Journal 9(4): 241–252.

Middleton P, Flaxel A & Cookson A (2005) Lean software management case study: Timberline inc. In: Anonymous Extreme Programming and Agile Processes in Software Engineering, Springer: 1–9.

Middleton P & Sutton J (2005) Lean software strategies: proven techniques for managers and developers, Productivity Pr.

Middleton P & Joyce D (2012) Lean software management: BBC Worldwide case study. Engineering Management, IEEE Transactions on 59(1): 20–32.

Morgan JM & Liker JK (2006) The Toyota product development system, Productivity press New York.

Müller, R. M., & Thoring, K. (2012). Design thinking vs. lean startup: A comparison of two user-driven innovation strategies. Leading Through Design, 151.

Münch J, Armbrust O, Kowalczyk M & Soto M (2012) Software Process Definition and Management, Springer.

Nagel RN & Dove R (1991) 21st century manufacturing enterprise strategy: An industry-led view, DIANE Publishing.

Naylor J. B., Naim M.M. & Berry D. (1999) Leagility: integrating the lean and agile manufacturing paradigms in the total supply chain. Int J Prod Econ 62(1): 107–118.

Neely, S., Stolt, S. (2013). Continuous delivery? easy! just change everything (well, maybe it is not that easy). In: Agile Conference (AGILE), 2013. IEEE, pp. 121–128.

Nguyen-Duc, A., Seppänen, P., & Abrahamsson, P. (2015). Hunter-gatherer cycle: a conceptual model of the evolution of software startups. In Proceedings of the 2015 International Conference on Software and System Process (pp. 199-203). ACM.

Nilsson, A., Bosch, J., Berger, C., 2014. Visualizing testing activities to support continuous integration: a multiple case study. In: Agile Processes in Software Engineering and Extreme Programming. Springer, pp. 171–186.

Nord RL, Ozkaya I & Sangwan RS (2012a) Making architecture visible to improve flow management in lean software development. Software, IEEE, 29(5): 33–39.

Nord, RL, Ozkaya, I, Kruchten, P, & Gonzalez-Rojas, M (2012b). In search of a metric for managing architectural technical debt. pp. 91–100.

Nybom, K., Smeds, J., & Porres, I. (2016). On the Impact of Mixing Responsibilities Between Devs and Ops (pp. 131–143). Springer, Cham

Ohno T (1988) Toyota production system: beyond large-scale production, Productivity press.

Oivo M, Birk A, Komi-Sirviö S, Kuvaja P & Solingen RV (1999) Establishing product process dependencies in SPI. In the Proceedings of European Software Engineering Process Group Conference.

Olsson, H. H., Alahyari, H., & Bosch, J. (2012). Climbing the "Stairway to Heaven" -- A Multiple-Case Study Exploring Barriers in the Transition from Agile Development towards Continuous Deployment of Software. In *38th Euromicro Conference on Software Engineering and Advanced Applications* (pp. 392–399). IEEE.

Olsson, H.H. , Bosch, J. , Alahyari, H. (2013). Towards R&D as innovation experiment systems: a framework for moving beyond agile software development. In: IASTED Multiconferences–Proceedings of the IASTED International Conference on Software Engineering, SE 2013, pp. 798–805.

Osterweil L (1987) Software processes are software too. Proceedings of the 9th international conference on Software Engineering, IEEE Computer Society Press: 2–13.

Palmer SR & Felsing M (2001) A practical guide to feature-driven development, Pearson Education.

Penners, R., & Dyck, A. (2015). Release Engineering vs. DevOps-An Approach to Define Both Terms. *Full-Scale Software Engineering*. Retrieved from https://www2.swc.rwth-aachen.de/docs/teaching/seminar2015/FsSE2015papers.pdf#page=53

Perepletchikov, M., Ryan, C., & Tari, Z. (2010). The impact of service cohesion on the analyzability of service-oriented software. IEEE Transactions on Services Computing, 3(2), 89-103.

Petersen K (2010) Implementing Lean and Agile software development in industry. Blekinge Institute of Technology. Doctoral Dissertation Series No. 2010:04

Poppendieck M & Poppendieck T (2003) Lean software development: An agile toolkit, Addison-Wesley Professional.

Poppendieck M & Poppendieck T (2006) Implementing lean software development: From concept to cash, Addison-Wesley Professional.

Poppendieck M & Poppendieck T (2009) Leading lean software development: Results are not the point, Pearson Education.

Poppendieck M & Cusumano MA (2012) Lean Software Development: A Tutorial. IEEE Software 29(5): 26–32.

Poppendieck, M. & Poppendieck, T. (2013). *The Lean Mindset: Ask the Right Questions*. Pearson Education.

Port, D., & Bui, T. (2009). Simulating mixed agile and plan-based requirements prioritization strategies: proof-of-concept and practical implications. European Journal of Information Systems, 18(4), 317-331.

Power, K. (2013). Understanding the impact of technical debt on the capacity and velocity of teams and organizations: Viewing team and organization capacity as a portfolio of real options. In *Managing Technical Debt (MTD), 2013 4th International Workshop on* (pp. 28-31). IEEE.

Rakitin S (2001) Manifesto elicits cynicism. IEEE Computer 34(12): 4.

Raman S (1998) Lean software development: Is it feasible? Digital Avionics Systems Conference, 1998. Proceedings, 17th DASC. The AIAA/IEEE/SAE, IEEE 1: C13/1-C13/8 vol. 1.

Rautonen, T. (2013). DevOps – Sovelluskehittäjän roolin evoluutio. https://gofore.com/devops-sovelluskehittajan-roolin-evoluutio/ (Retrieved June 30, 2017)

Reifer, D. J. (2002). How good are agile methods?. IEEE software, 19(4), 16-18.

Reinertsen DG (2009) The principles of product development flow: second generation lean product development, Celeritas Redondo Beach, Canada.

Ries, E (2011). The lean startup: How today's entrepreneurs use continuous innovation to create radically successful businesses. Random House LLC

Rivero, J. M., Grigera, J., Rossi, G., Luna, E. R., Montero, F., & Gaedke, M. (2014). Mockup-driven development: providing agile support for model-driven web engineering. Information and Software Technology, 56(6), 670-687.

Rodríguez P, Markkula J, Oivo,M, & Turula K (2012). Survey on agile and lean usage in Finnish software industry. In Empirical Software Engineering and Measurement (ESEM), 2012 ACM-IEEE International Symposium on (pp. 139-148). IEEE.

Rodríguez, P., Mikkonen, K., Kuvaja, P., Oivo, M., & Garbajosa, J. (2013). Building lean thinking in a telecom software development organization: strengths and challenges. In Proceedings of the 2013 international conference on software and system process (pp. 98-107). ACM.

Rodríguez, P., Partanen, J., Kuvaja, P., & Oivo, M. (2014). Combining lean thinking and agile methods for software development: A case study of a finnish provider of wireless embedded systems detailed. In System Sciences (HICSS), 2014 47th Hawaii International Conference on (pp. 4770-4779). IEEE.

Rodríguez, P., Haghighatkhah, A., Lwakatare, L. E., Teppola, S., Suomalainen, T., Eskeli, J., Karvonen, T., Kuvaja, P., Verner, J.M. and Oivo, M.,(2017) Continuous deployment of software intensive products and services: A systematic mapping study. Journal of Systems and Software, 123, 263-291.

Rosenberg D & Stephens M (2003) Extreme programming refactored: the case against XP, Apress.

Royce WW (1970) Managing the development of large software systems. Proceedings of IEEE WESCON, Los Angeles, CA, USA: 1–9.

Schneider, T. (2016). Achieving Cloud Scalability with Microservices and DevOps in the Connected Car Domain. In CEUR workshop proceedings on continuous software engineering (pp. 138–141). CEUR-WS.org. Retrieved from http://ceur-ws.org/Vol-1559/

Schwaber, K., & Beedle, M. (2002) Agile software development with Scrum, Prentice Hall Upper Saddle River.

Schwaber, K., & Sutherland, J. (2016) The Scrum Guide. The definitive guide to Scrum: the rules of the game, http://www.scrumguides.org/ (last accessed 09.08.2017)

Scott, E., Rodríguez, G., Soria, Á., & Campo, M. (2014). Are learning styles useful indicators to discover how students use Scrum for the first time?. Computers in Human Behavior, 36, 56-64.

Sedano, T., Ralph, P., & Péraire, C. (2017). Software development waste. In *Proceedings of the 39th International Conference on Software Engineering* (pp. 130-140).

Seppänen, P., Tripathi, N., Oivo, M & Liukkunen, K. (2017). How Are Product Ideas Validated? The Process from Innovation to Requrement Engineering in Software Startups. In 8th International Conference on Software Business (ICSOB)

Seppänen, P., Oivo, M & Liukkunen, K (2016). The initial team of a software startup, Narrow-shouldered innovation and broad-shouldered implementation. In To be published in 22nd ICE/IEEE International Technology Management Conference.

Shah R & Ward PT (2007) Defining and developing measures of lean production. J Oper Manage 25(4): 785–805.

Shahin, M. (2015). Architecting for devops and continuous deployment. In Proceedings of the ASWEC 2015 24th Australasian Software Engineering Conference (pp. 147-148). ACM.

Shahin, M., Zahedi, M., Babar, M. A., & Zhu, L. (2017). Adopting Continuous Delivery and Deployment: Impacts on Team Structures, Collaboration and Responsibilities. In 21st Evaluation and Assessment in Software Engineering Conference (EASE).

Sharifi H & Zhang Z (1999) A methodology for achieving agility in manufacturing organisations: An introduction. Int J Prod Econ 62(1): 7–22.

Sherehiy B, Karwowski W & Layer JK (2007) A review of enterprise agility: concepts, frameworks, and attributes. Int J Ind Ergonomics 37(5): 445–460.

Shull F. (2011) Perfectionists in a World of Finite Resources. IEEE Software.

Smeds, J., Nybom, K., & Porres, I. (2015). DevOps: A Definition and Perceived Adoption Impediments. In 16th International Conference on Agile Software Development (XP) (pp. 166–177). Helsinki: Springer International Publishing.

Sommerville I (2010) Software Engineering (9th Edition). Addison-Wesley.

Staats BR, Brunner DJ & Upton DM (2011) Lean principles, learning, and knowledge work: Evidence from a software services provider. J Oper Manage 29(5): 376–390.

Stapleton J (2003) DSDM: Business focused development, Addison-Wesley Professional.

Steinert, M., & Leifer, L. J. (2012). 'Finding One's Way': Re-Discovering a Hunter-Gatherer Model based on Wayfaring. International Journal of Engineering Education, 28(2), 251.

Sutherland, J. (2004). Agile development: Lessons learned from the first scrum. *Cutter Agile Project Management Advisory Service: Executive Update*, *5*(20), 1-4.

Sutherland, J., & Schwaber, K. (2011). The scrum papers: nut, bolts, and origins of an Agile framework. *SCRUM Training Institute*, *152*.

Takeuchi, H., & Nonaka, I. (1986). The new new product development game. *Harvard business review*, *64*(1), 137-146

Tang, C., Kooburat, T., Venkatachalam, P., Chander, A., Wen, Z., Narayanan, A., Karl, R. (2015). Holistic configuration management at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles - SOSP '15* (pp. 328–343).

Tessem, B., & Iden, J. (2008). Cooperation between developers and operations in software engineering projects. In Proceedings of the 2008 International Workshop on Cooperative and Human Aspects of Software Engineering.

Tierney J (1993) Eradicating mistakes from your software process through Poka Yoke. Proc.6th Inte.Software Quality Week : 300–307

Tokatli N (2008) Global sourcing: insights from the global clothing industry—the case of Zara, a fast fashion retailer. Journal of Economic Geography 8(1): 21–38.

Tom, E., Aurum, A., & Vidgen, R. (2013). An exploration of technical debt. *Journal of Systems and Software*, *86*(6), 1498-1516.

Trimble J & Webster C (2013) From Traditional, to Lean, to Agile Development: Finding the Optimal Software Engineering Cycle. System Sciences (HICSS), 2013 46th Hawaii International Conference on, IEEE: 4826–4833.

Turk D, France R & Rumpe B (2002) Limitations of agile software processes. Third International Conference on eXtreme Programming and Agile Processes in Software Engineering (XP 2002): 43–46.

Vähäniitty, J., & Rautiainen, K. T. (2008). Towards a conceptual framework and tool support for linking long-term product and business planning with agile software development. In Proceedings of the 1st international workshop on Software development governance (pp. 25-28). ACM.

Van Hoek RI (2000) The thesis of leagility revisited. International Journal of Agile Management Systems 2(3): 196–201.

VerisonOne, Inc (2016) 10th annual "state of agile development" survey. URL https://versionone.com/pdf/ VersionOne-10th-Annual-State-of-Agile-Report.pdf

Vilkki K (2010) When agile is not enough. In: Lean Enterprise Software and Systems, Springer: 44–47.

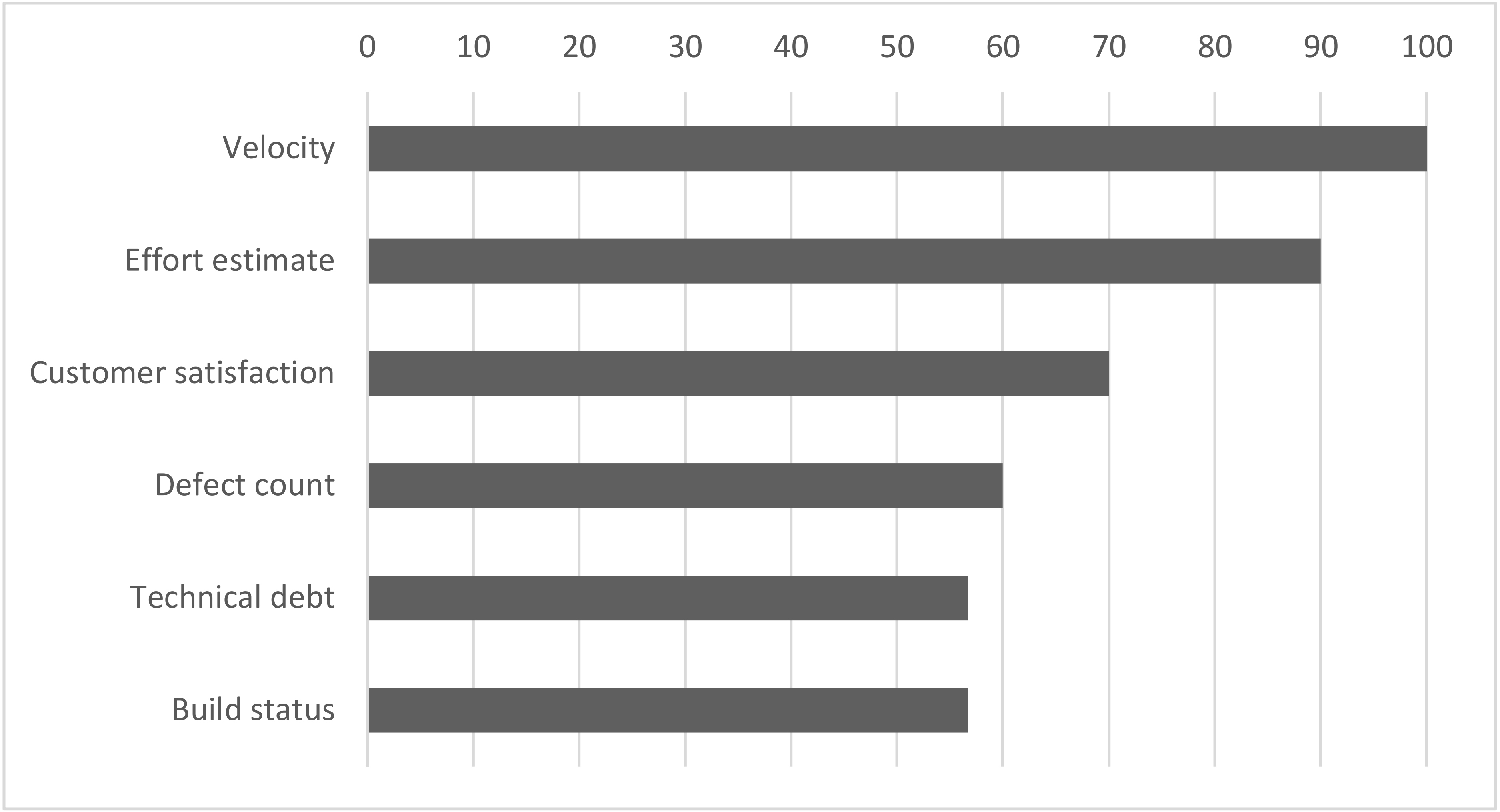
Vilkki K & Erdogmus H (2012) Point/Counterpoint. IEEE Software 29(5): 60–63.

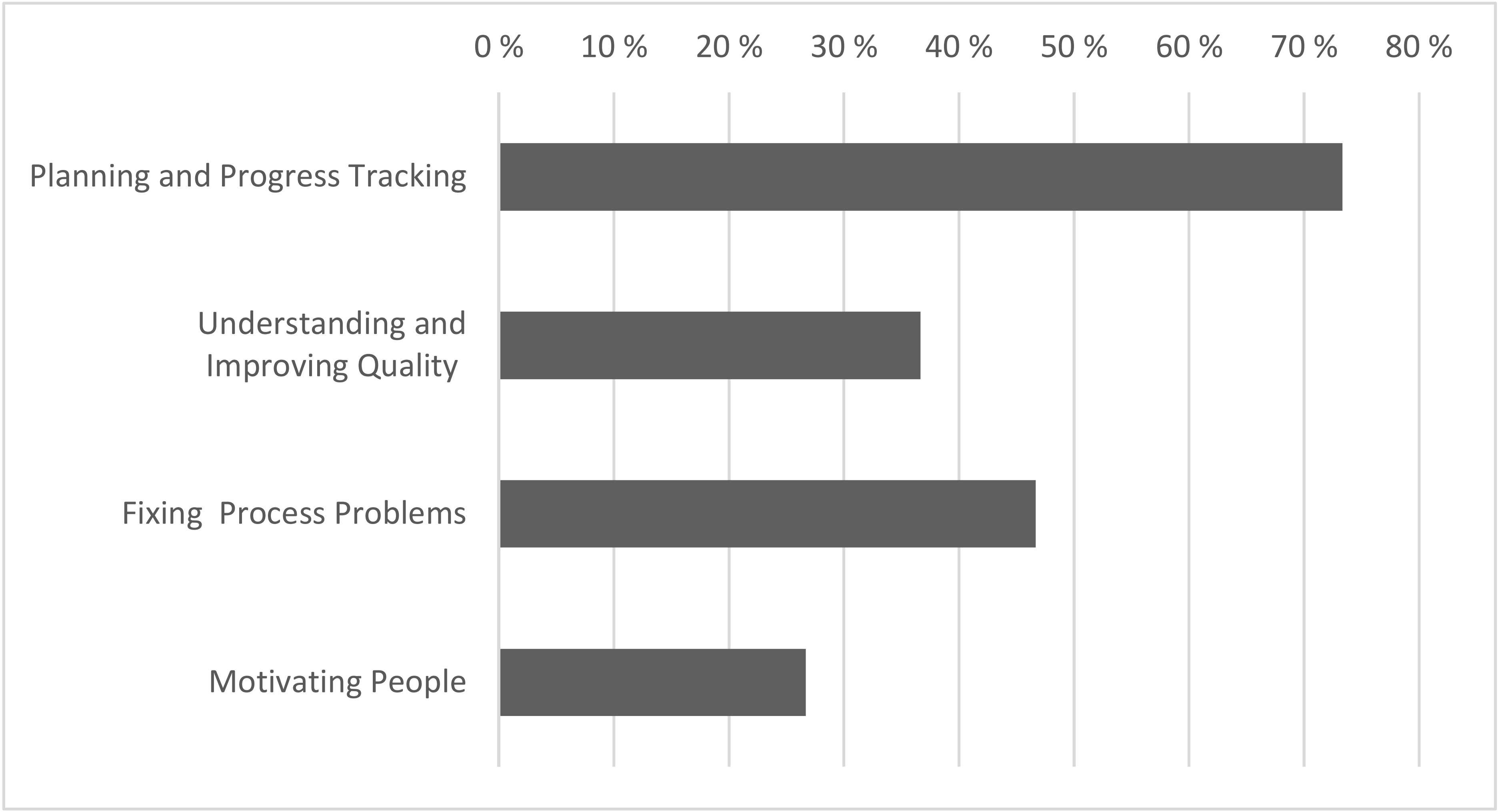
Wang X, Conboy K & Cawley O (2012) 'Leagile' software development: An experience report analysis of the application of lean approaches in agile software development. J Syst Software 85(6): 1287–1299.

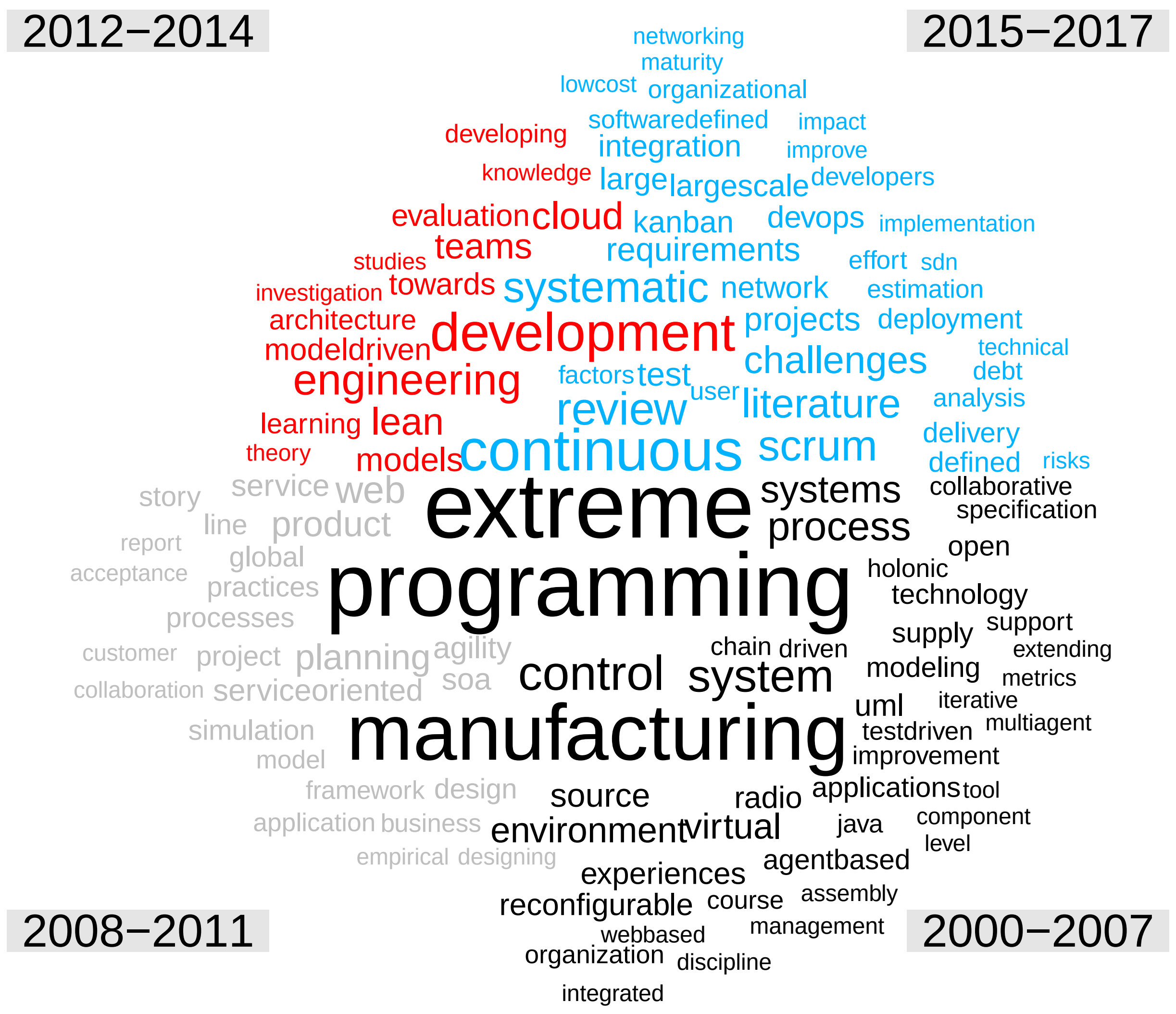
Wasserman T (2013) Low ceremony processes for short lifecycle projects. Keynote at the 2013 International Conference on Software and System Process, ACM.

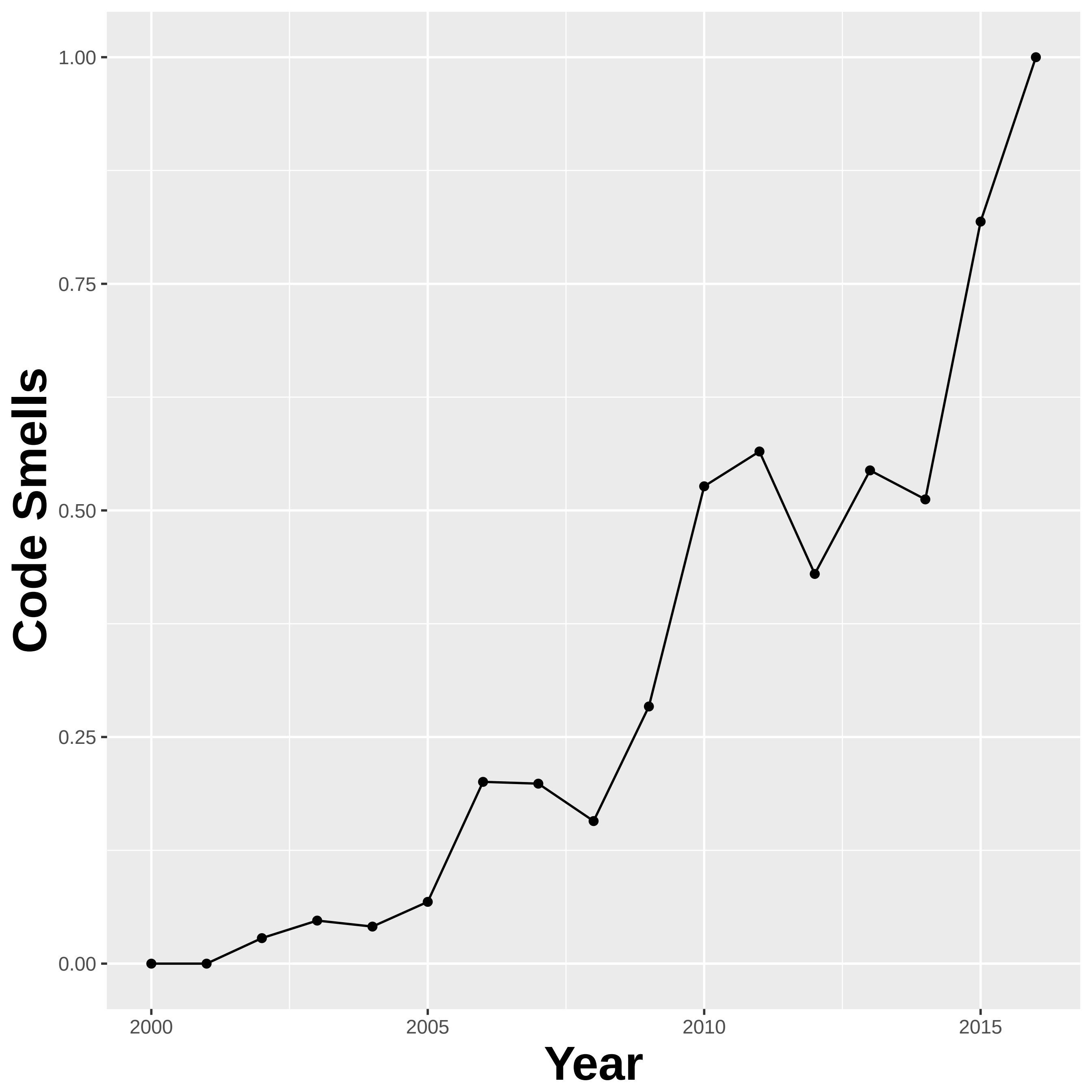
Womack JP, Jones DT & Roos D (1990) The Machine That Changed the World: The Story of Lean Production: How Japan's Secret Weapon in the Global Auto Wars Will Revolutionize Western Industry. New York, NY: Rawson Associates.

Womack JP & Jones DT (1996) Lean thinking: Banish waste and create wealth in your organisation. Rawson Associates, New York.

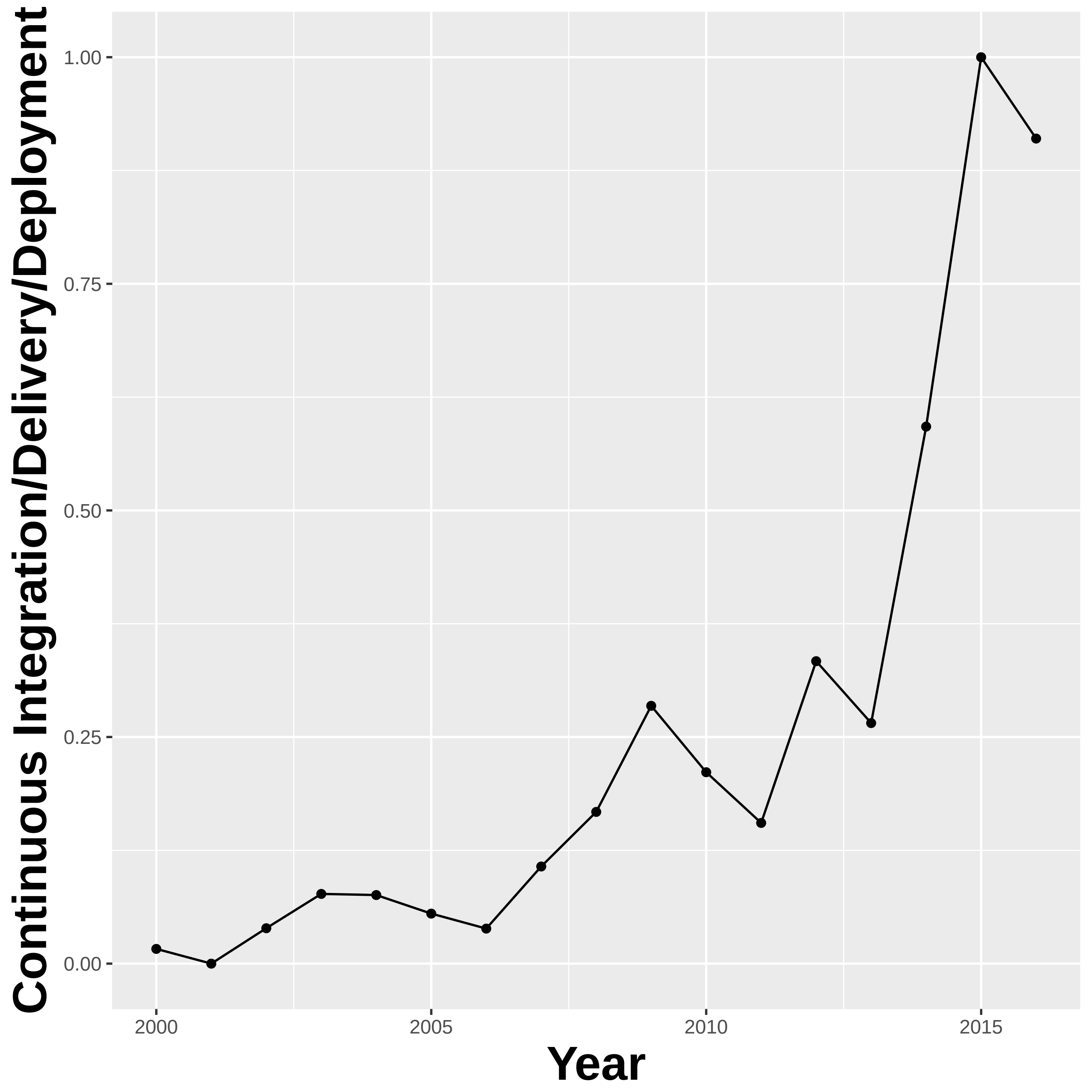

0
10
20
30
40
50
60
70
80
90
100
Velocity
Effort estimate
Customer satisfaction
Defect count
Technical debt
Build status


0 %
10 %
20 %
30 %
40 %
50 %
60 %
70 %
80 %
Planning and Progress Tracking
Understanding and Improving Quality
Fixing Process Problems
Motivating People

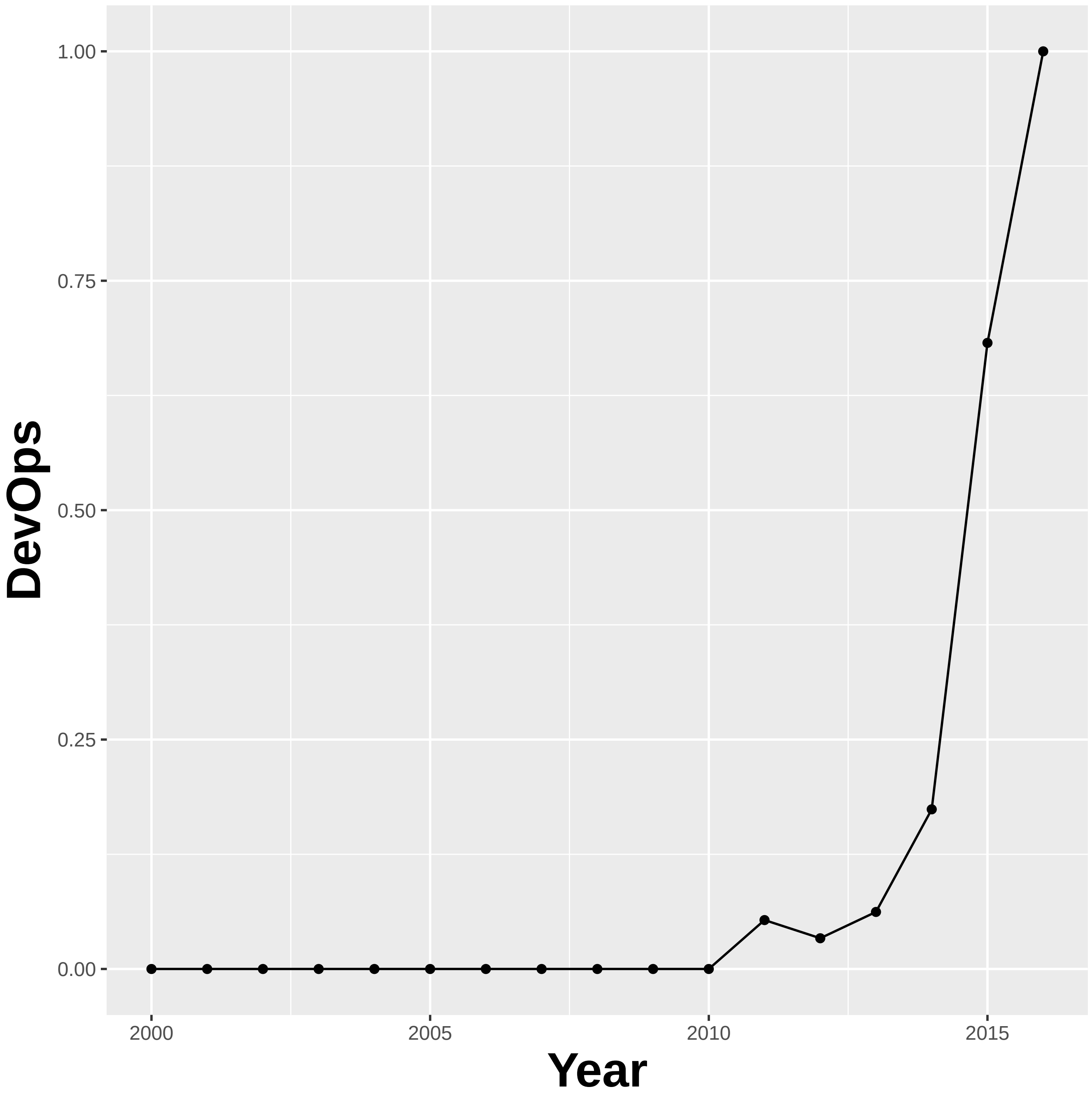

2012−2014
2015−2017
2008−2011
2000−2007
networking
maturity
lowcost
organizational
developing
softwaredefined
impact
integration
improve
knowledge
large
largescale
developers
evaluation
cloud
kanban
devops
implementation
teams
requirements
studies
effort
sdn
investigation
towards
systematic
network
estimation
architecture
development
projects
deployment
modeldriven
technical
challenges
debt
engineering
factors
test
user
literature
analysis
learning
lean
review
delivery
theory
models
continuous
scrum
defined
risks
story
service
web
extreme
systems
collaborative
specification
line
product
process
report
global
open
acceptance
practices
programming
holonic
technology
processes
supply
support
customer
project
planning
agility
chain
driven
extending
control
system
modeling
metrics
collaboration
serviceoriented
soa
uml
iterative
simulation
manufacturing
testdriven
multiagent
model
improvement
framework
design
source
radio
applications
tool
application
business
environment
virtual
java
component
level
empirical
designing
experiences
agentbased
reconfigurable
course
assembly
webbased
management
organization
discipline
integrated



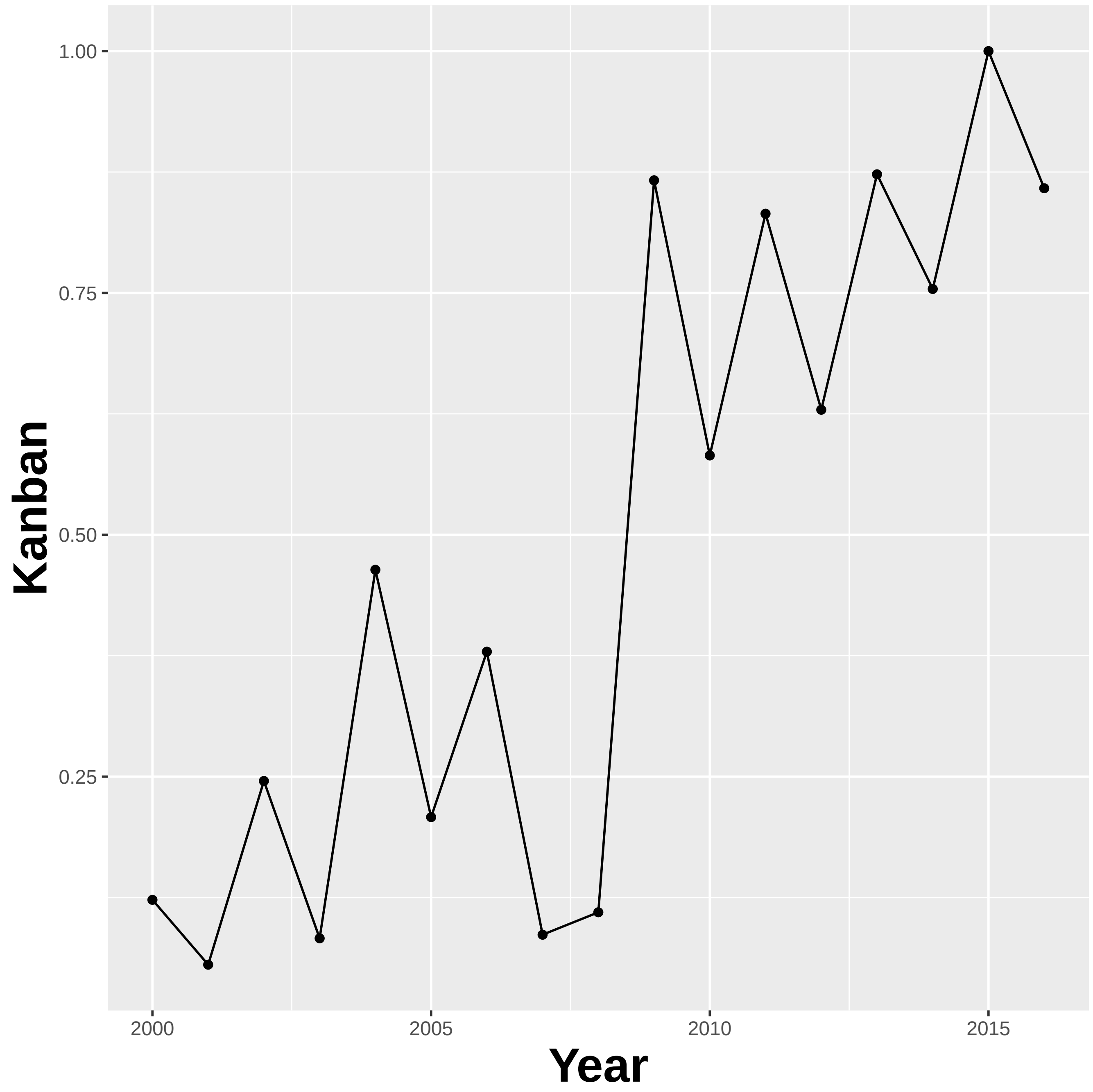
Code Smells
1.00
0.75
0.50
0.25
0.00
2000
2005
2010
2015
Year



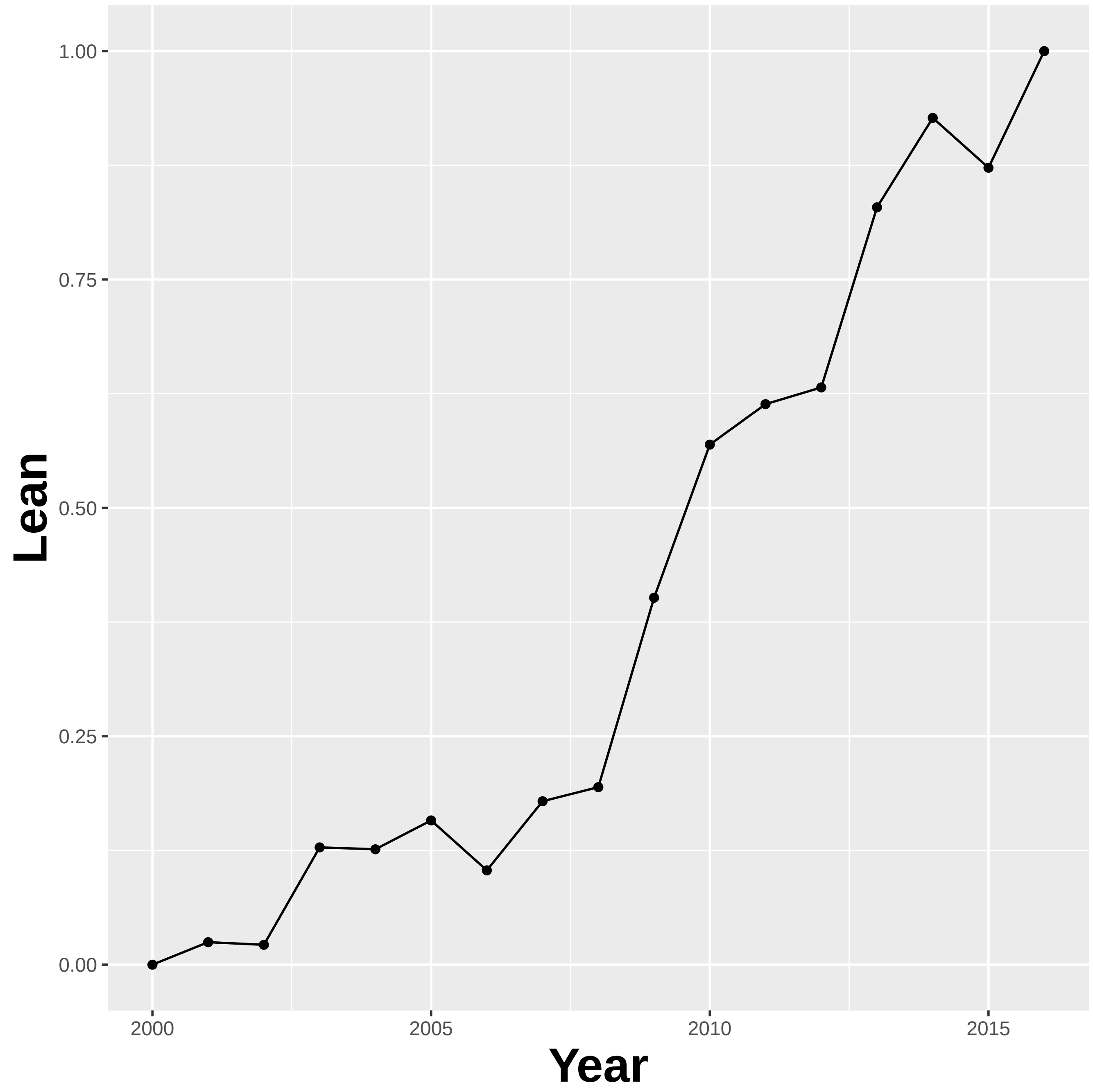
Continuous Integration/Delivery/Deployment
1.00
0.75
0.50
0.25
0.00
2000
2005
2010
2015
Year



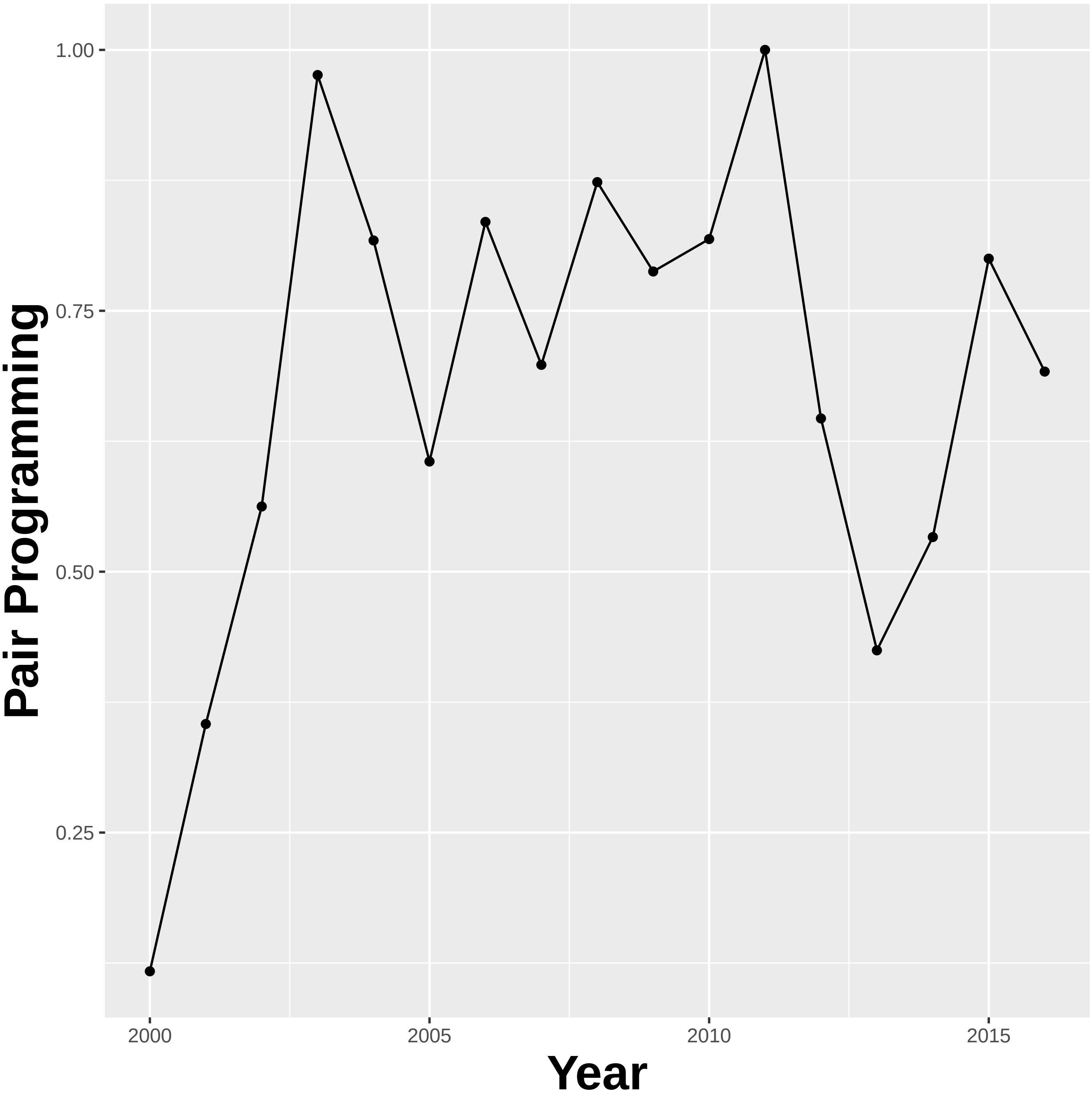
DevOps
1.00
0.75
0.50
0.25
0.00
2000
2005
2010
2015
Year



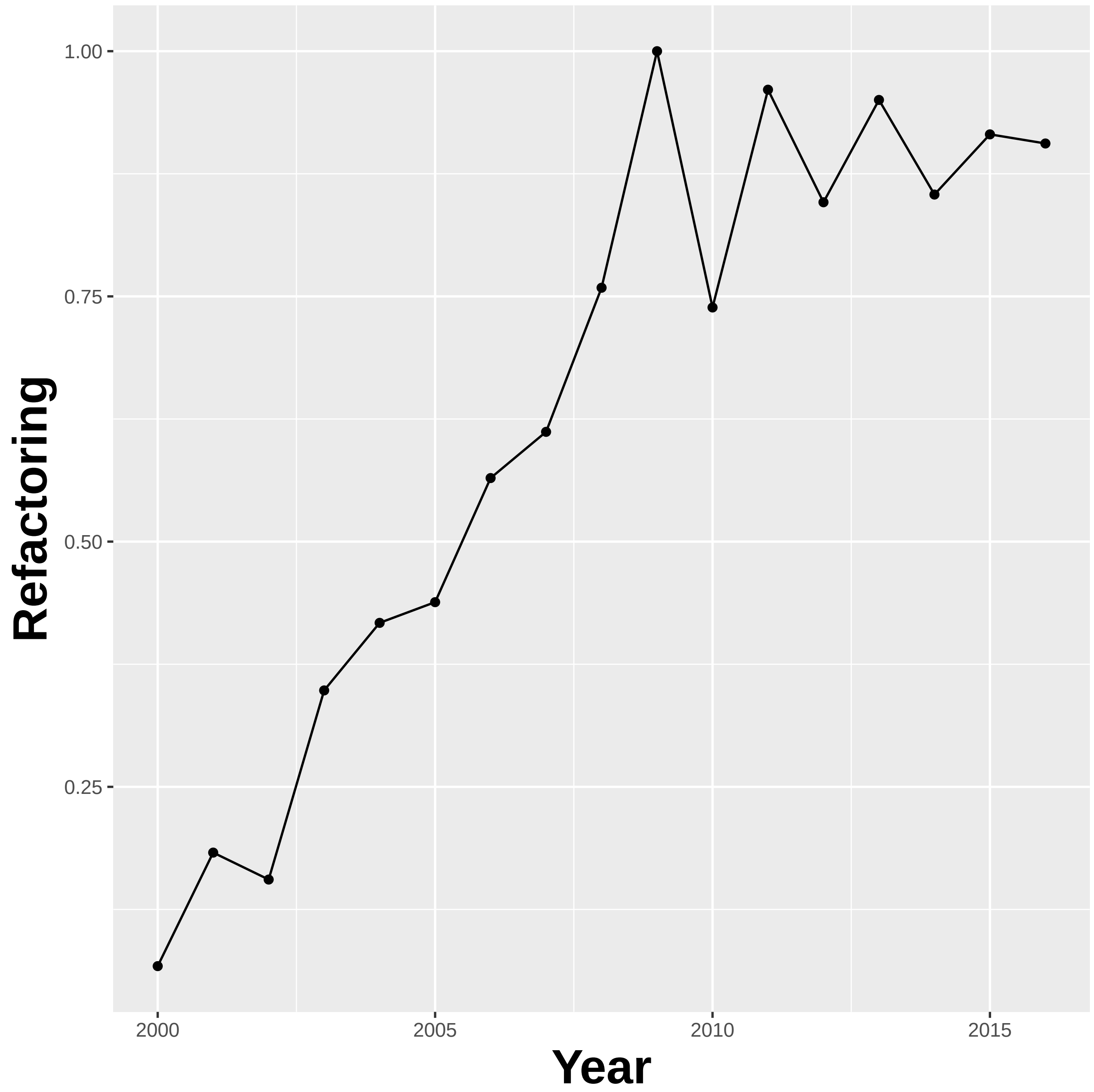
1.00
0.75
0.50
0.25
Kanban
2000
2005
2010
2015
Year



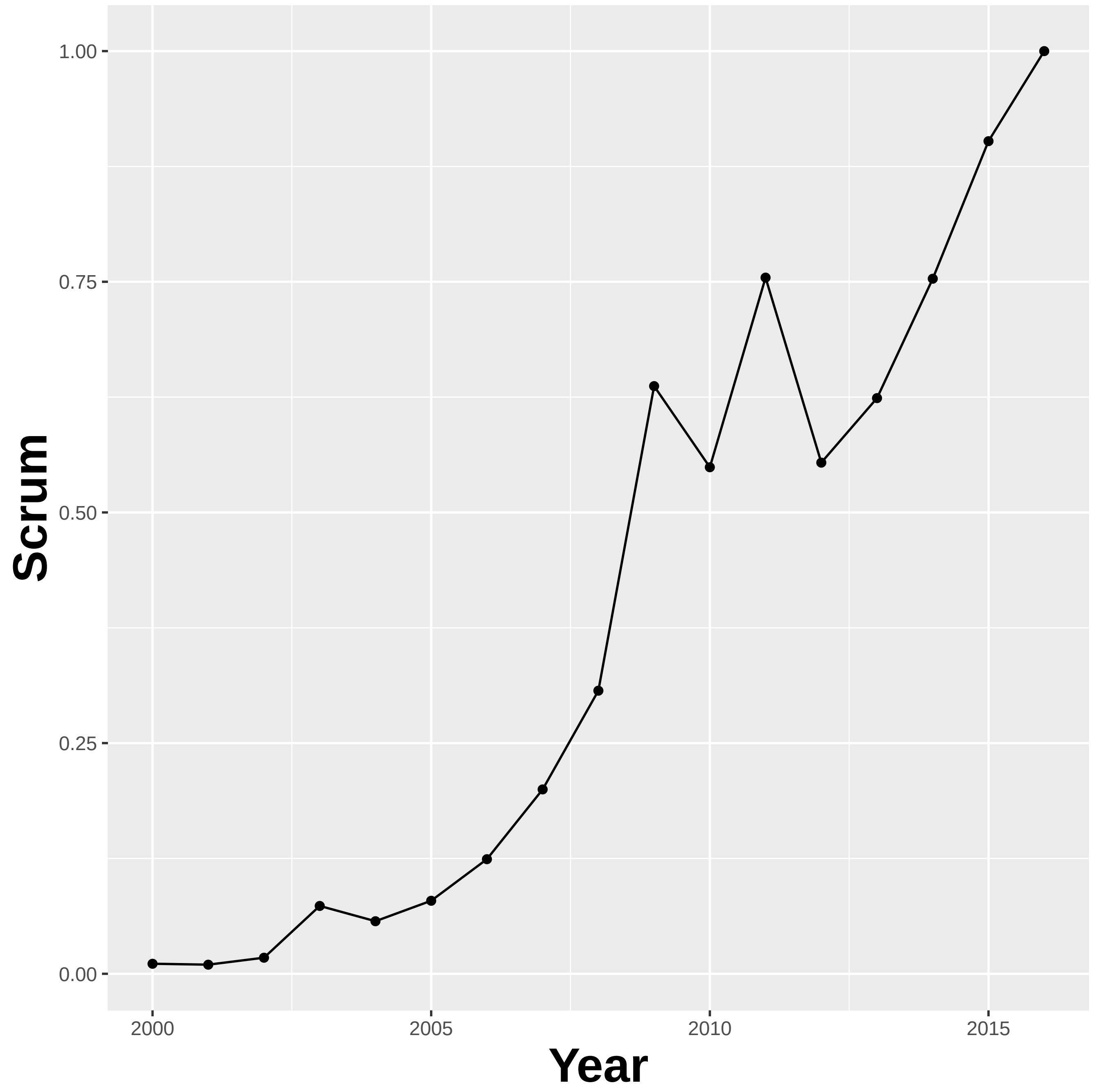
Lean
1.00
0.75
0.50
0.25
0.00
2000
2005
2010
2015
Year



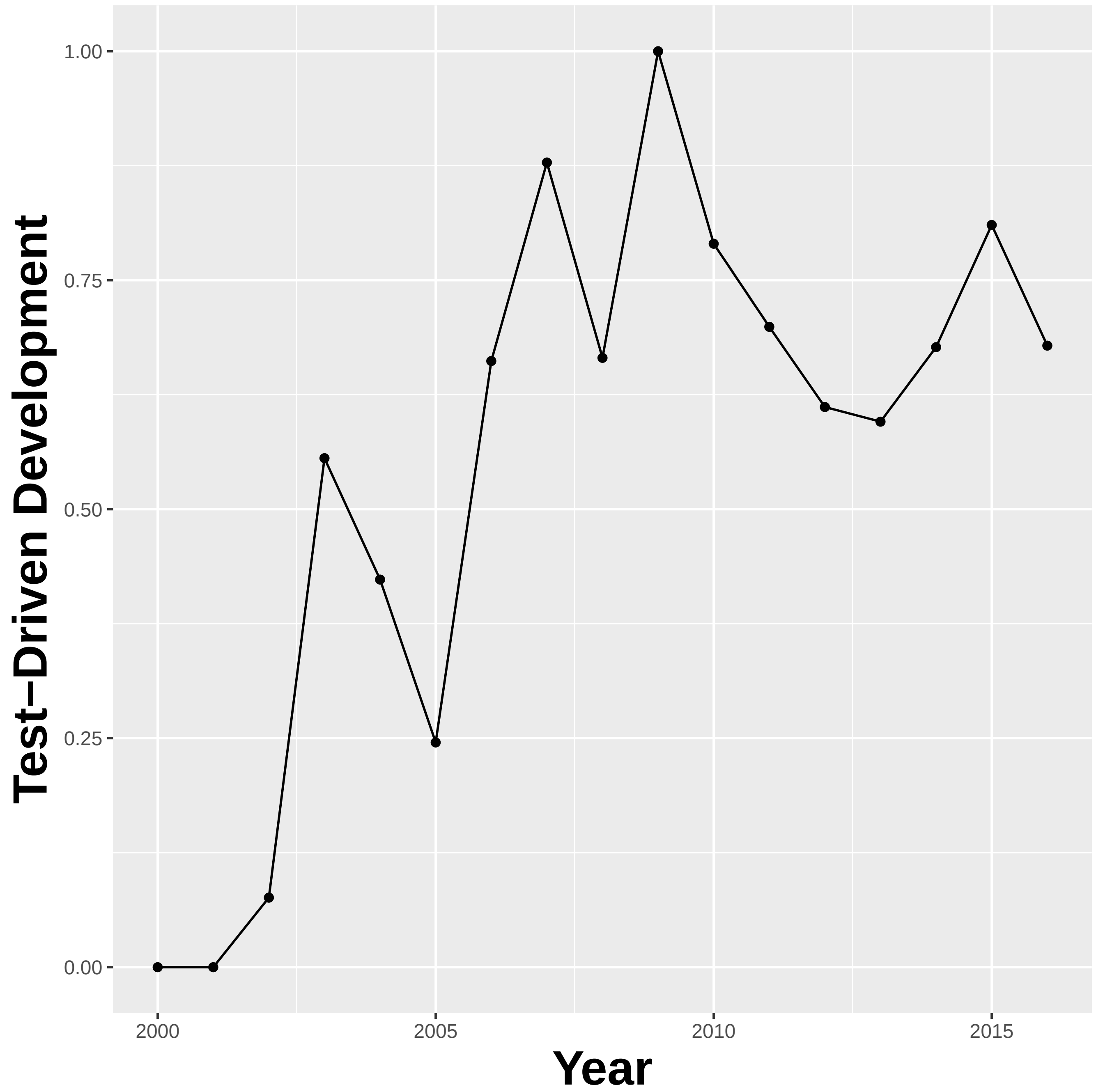
Pair Programming
1.00
0.75
0.50
0.25
2000
2005
2010
2015
Year



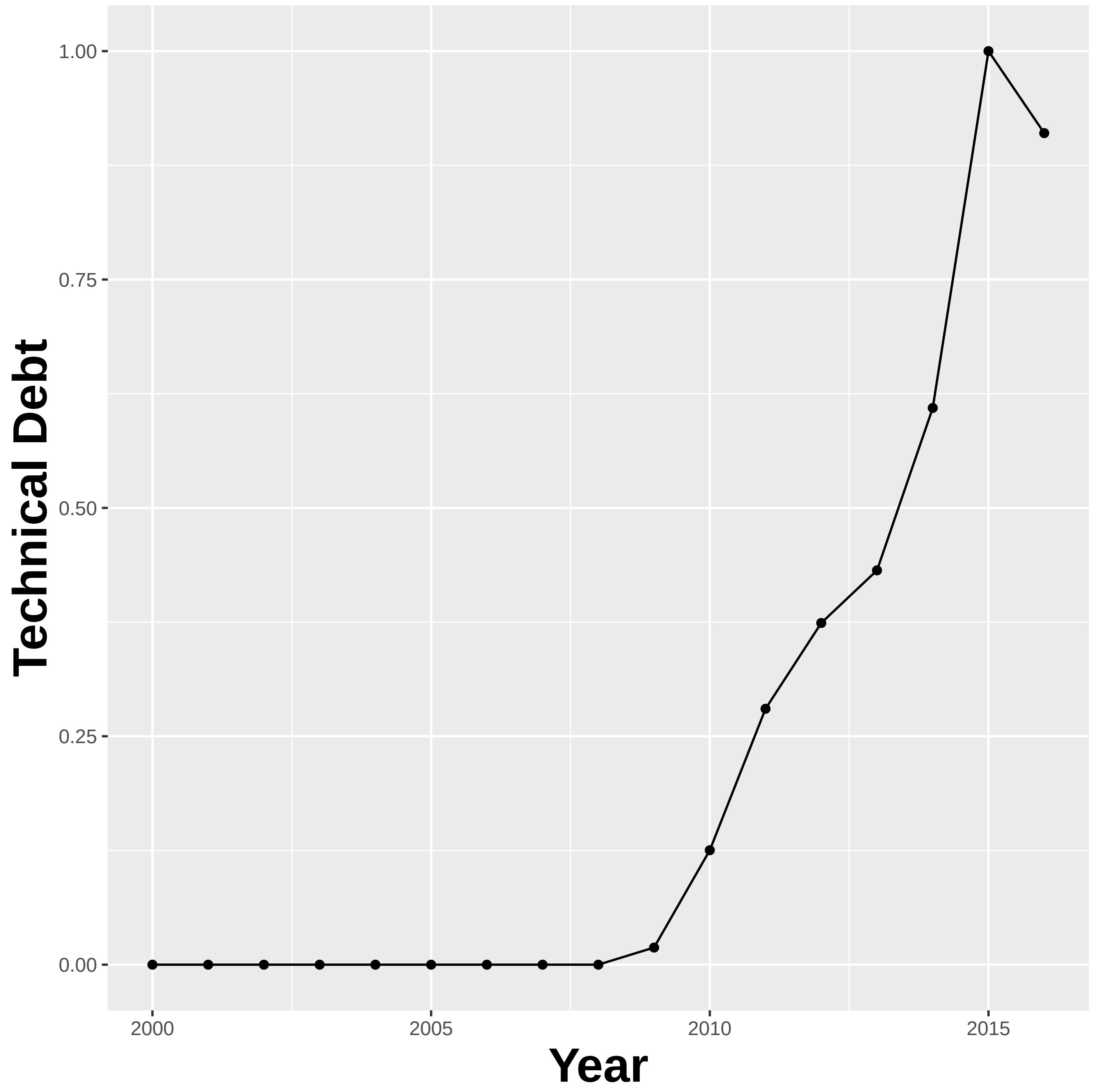
Refactoring
1.00
0.75
0.50
0.25
2000
2005
2010
2015
Year



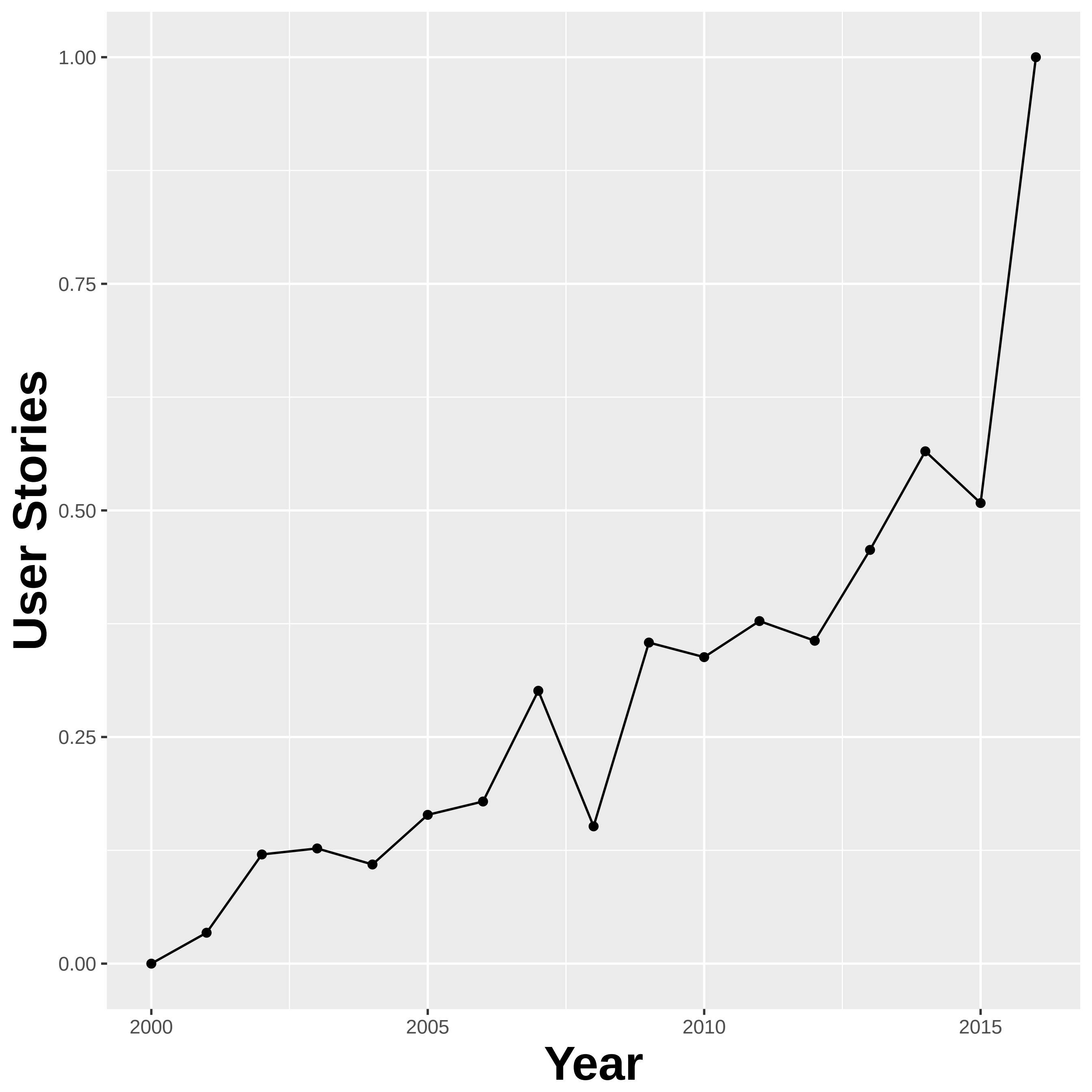
Scrum
1.00
0.75
0.50
0.25
0.00
2000
2005
2010
2015
Year

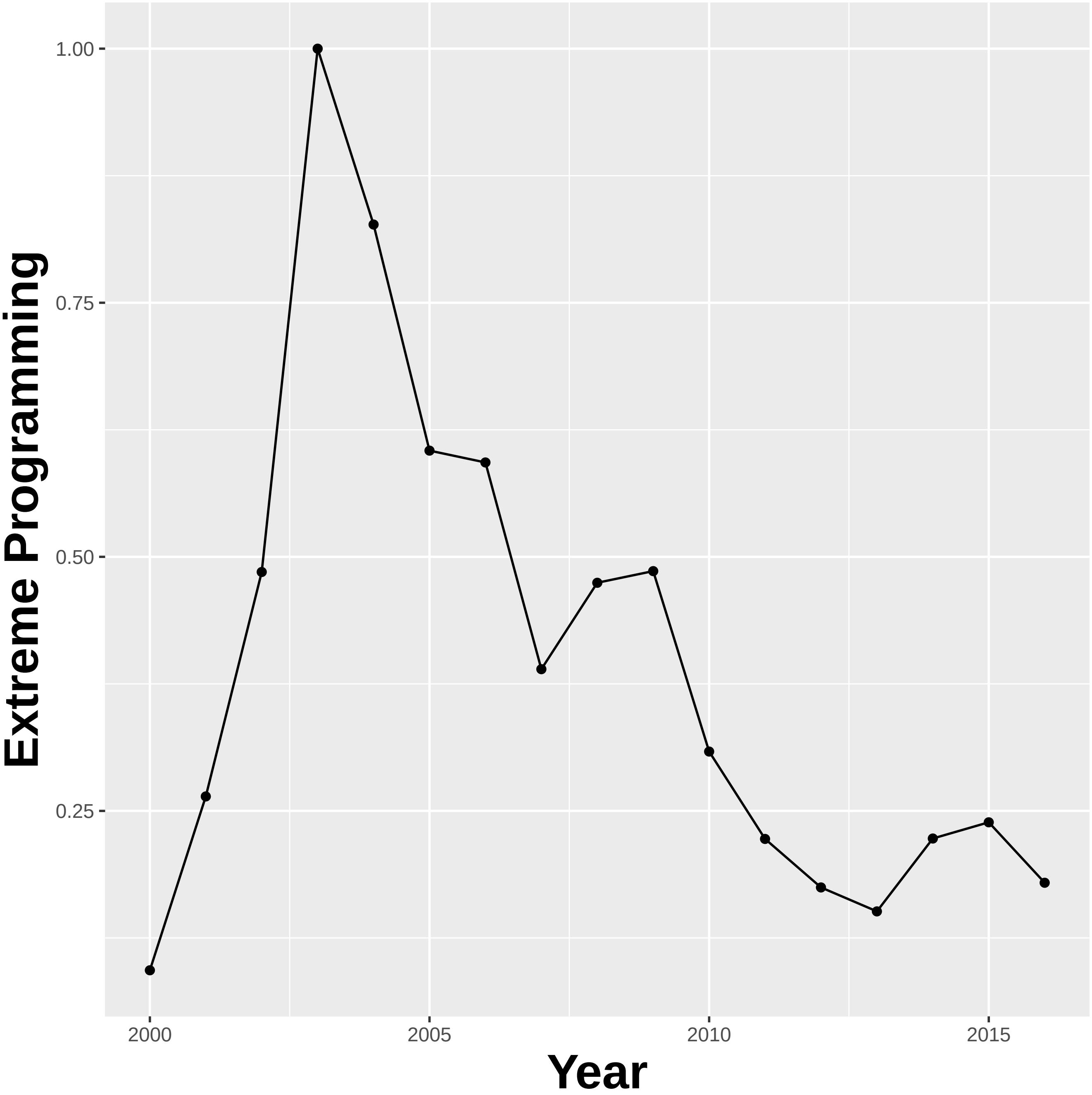


Test–Driven Development
1.00
0.75
0.50
0.25
0.00
2000
2005
2010
2015
Year



Technical Debt
1.00
0.75
0.50
0.25
0.00
2000
2005
2010
2015
Year


User Stories
1.00
0.75
0.50
0.25
0.00
2000
2005
2010
2015
Year



Extreme Programming
1.00
0.75
0.50
0.25
2000
2005
2010
2015
Year