

Chapter 2

Agile Software Development

Software is nowadays omnipresent in many consumer and business products. From cars to complex business software over simple smart phone apps, many people regularly use software-based products or services at home or at work. With more than two billion people using broadband internet today, new markets emerge for software companies challenging established market incumbents with software-based services and products. Software brings life to various types of computer systems which have steadily become cheaper, more powerful, more compact, and more energy efficient over the last decades.¹ Consequently, these systems can be integrated into various traditional products to provide diversifying functionality (MacCormack et al. 2001). Andreessen (2011) summarized the trend succinctly: “Software is eating the world”.

From a technical point of view, *software* is a “set of programs, procedures, and routines which are associated with the operation of a computer system”.² Since the first days of programmable machines, both the computer systems and development processes have evolved dramatically. The first software programs were written by electrical engineers and mathematicians in laboratories for military applications in the 1940s. Software development was a very experimental work, with no supporting tools or experience existing. The developed software provided customized functionality for specific purposes. As the hardware and the development process was very expensive, only few applications justified the high expenditure.

Today, the *software industry* is a global industry with an estimated revenue of more than \$400 billion per year.³ More than half of the revenue is generated by the top ten software vendors, such as Microsoft, Oracle, IBM, or SAP. But

¹Computing power doubled every 2 years (Moore’s Law), while energy efficiency double every 1.8 years (Kooimey’s law) (Kooimey et al. 2011; Moore 1965).

²<http://www.merriam-webster.com/dictionary/software>.

³<http://www.gartner.com/newsroom/id/2696317>.

there are more and more small companies providing complementary software products as entry barriers to the software market are ever decreasing (Kude et al. 2012). Small start-up companies can develop and offer software products to large markets at limited costs due to the ubiquitous availability of computers, standardized programming languages, interfaces, as well as reusable software packages with free development tools and supportive development communities. Even amateurs contribute to large-scale software development projects as everybody can develop code for open source communities and participate in the development process (Setia et al. 2012).

The *software engineering* discipline offers solutions to the practical problems of developing software (Sommerville 2004). Today, software engineering is an independent profession and an engineering discipline embedded in computer science and traditional engineering disciplines. It deals with all activities required to economically solve real world problems with reliable software programs. It helps developers with “the application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software” (IEEE 1990) and encompasses the application of various theories, methods, processes and tools for the development and maintenance of software within financial and organizational constraints of software development projects.

This chapter first discusses the complex nature of software development projects, to then describe different software development approaches suggested by the software engineering discipline. Three different generations of software development processes are briefly introduced and discussed. *Agile software development* has gained much popularity over the last 10 years motivating this study. The chapter concludes with a brief overview of the practitioner and research literature on agile software development.

2.1 Software Development

Software development is a complex process and many software development projects end in delayed projects results, failure to meet the original project goals, or are even entire project cancellations (Standish 2014). This high failure rate is surprising given the history of more than 60 years of software development projects and numerous best-practices books, studies about project failures, and countless development tools, methods, and processes.

There is a long list of failed project failures. In 2004, for example, one of Britain’s biggest retailer, Sainsbury, had to cancel a \$525 million IT project intended to automate the supply chain of the company. In the end, the failed system caused the company to hire an additional 3000 people to manually record all its stock items.⁴

⁴<http://www.computerweekly.com/news/2240058411/Sainsburys-writes-off-260m-as-supply-chain-IT-trouble-hits-profit>.

In 2013, the United States government started the website healthcare.org intended to process thousands of users simultaneously. The IT project had been outsourced to several contractors for more than \$300 million (Brill 2014). When launched, the information system was not able to adequately process the incoming requests. As a consequence, the project meant a tremendous disaster for the government as it was at the center of a media debate for several weeks.

Project failure is common in numerous industries. However, these are just two examples of a long list of failed software development projects (Charette 2005) and it is particular to software development projects that much money can be spent for no result. The periodical reports of the Standish Group Inc., which examined more than 50,000 IT projects, shows that about a third of IT development projects are canceled before completion. Moreover, about half of the projects will cost almost twice as initially projected. Royce et al. (2009) concluded, there is hardly another field in engineering where such a poor success rate is considered normal, much less tolerated. The economic damage, however, is significant (Standish 2014) which is why analyzing these project failures is not purely an academic exercise. Instead, there is a strong incentive for organizations to understand the reasons for failure and determinants of success.

Despite a history of IT project failures, research on success factors of software development projects remains limited. Sambamurthy and Kirsch (2000) concluded in their literature review article that “deficiencies exist in the knowledge about the effective management of complex systems development processes”. Until today, there are no final answers on the successful management of software development projects.

2.1.1 Software Development Complexity

There are several sources of complexity which may explain the high failure rates of software development projects. Reasons include the fundamental characteristics of software, the contexts in which software is used, the complexity of the software development tasks, and the general nature of software development projects. The following sections elaborate on these aspects.

Software Complexity

Software is an intangible good which consist of small functional, closely interlinked modules or components. As these components grow in numbers, it becomes increasingly difficult to keep track and see how they work together, even for the most talented programmers. Most software components can have different possible inputs, outputs, states or dependencies. It is therefore nearly impossible to test all combinations. As testing software comes at a cost, it is often impossible to consider all interactions of a program, even for small programs. Moreover, software

inconsistencies may be detrimental as changing one part of the program might have consequences for the functionality of the entire software program. For these reasons, it can be extremely difficult to find errors, measure the quality of a software program, or to properly assess its full functional spectrum. This makes software development so difficult.

The technological progress has provided software developers with almost unlimited computational power, memory, and storage capacity, and a global connectivity of devices. Therefore, software is hardly constrained by physical laws, but only limited to human creativity (Lee 1999). On the one hand, this gives developers a considerable freedom of design. On the other hand, it may add to the complexity of today's software programs.

In addition, the size of the software packages has constantly grown during the last decades. Existing software libraries and standardized software packages from third party providers are integrated into new product development (Spinellis and Giannikas 2012) as software functionality can be reused. This leads to an increasing size of the developed software packages quickly exceeding the cognitive capabilities of developers. For instance, the first operating systems of personal computers, still one of the largest software applications, included between 4–5 million lines of code in the beginning of the 1990s. The latest operating systems are implemented with about 50 million lines of code (Maraia 2006). A modern car is estimated to run with about 100 million lines of code (Charette 2009).

The reuse of software functionality adds to the complexity, as the usage of existing software functionality in unintended environments may cause unpredicted side effects. Brooks (1995) even stated “software entities are more complex for their size than any other human construct” as software is invisible, unvisualizable, and subject to continuous change.

Software Development Task Complexity

Literature has identified three dimensions of task characteristics that constitute task complexity (Wood 1986): *component*, *coordinative*, and *dynamic complexity* (Campbell 1988). All three have been found to be inherent in software development tasks.

Software development includes various subtasks such as programming, testing, project management, documentation, requirements specification, installation of development tools, and communication with colleagues or other project stakeholders (Begel and Simon 2010). To succeed, developers need to combine organizational, technical, as well as social skills. Before writing the first line of code, analysis of the software requirements is one of the most critical activities (Hickey and Davis 2004). It requires a solid understanding of the application problem domain and technical expertise. For most projects, this knowledge is dispersed amongst different stakeholders. Mostly, requirements are neither clearly documented nor concisely specified by the users who often only know what they expect upon seeing the final product. Even if requirements can be fully specified, software development remains

a non-deterministic activity with no single identifiable path on how to implement a program (Sommerville 2004).

Component complexity is high when a task requires several acts and various information cues to be accomplished. Every software development task involves several sub-tasks to be composed. Moreover, as software is highly modularized, software developers need to consider different components and their interdependence. Both aspects lead to high task complexity. Second, software development has a high *coordinative complexity*. Coordinative complexity is low if a linear function of task inputs leads to the task product. In software development, there are many ways to implement requirements. The inputs of software development tasks are intangible ideas that need to be interpreted. Mostly, there is no single correct way but several options to write a software program. Finally, *dynamic complexity* is caused by changes of the task environments. Software development tasks are often subject to change, e.g. the used technology may change during the development process. Oftentimes, the software requirements change as they are either not clearly defined, misunderstood, or need to be redefined. Overall, software development is a highly complex task with component, coordinative, and dynamic complexity. All three are a central sources of complexity in software development projects.

Software Development Project Complexity

Software development projects are described as inherently complex as they must deal with both technological and organizational factors, often outside the control of the project team. Organizational complexity increases with the number of specialized units as well as the number of relationships between them (Baccarini 1996). Moreover, uncertain environments due to unpredictable markets, changing customer requirements, pressure to shorter time-to-market cycles, and rapidly changing information technologies (Baskerville et al. 2001) can add to it. Many software systems are very large and thus beyond the ability of a single software developer to build, leading to the division of labor amongst different contributors. Many projects include actors from diverse geographic, organizational, and social backgrounds (Dibbern et al. 2004) which increases organizational complexity in software development projects (Kraut and Streeter 1995).

Besides organizational complexity, software development projects mostly face technological complexity (Schmidt et al. 2001). Technological complexity refers to the number of and relationship between inputs, outputs, tasks, and technologies. The previously discussed characteristics of software contribute to technological complexity of a software development project.

Rapid evolution of technology and new product opportunities lead to change and uncertainty as an inherent characteristic of software development projects (Madsen 2007). Uncertainty is broadly defined as the absence of complete information about an organizational phenomenon being studied (Argote 1982). It leads to an inability to accurately predict the project progress (Dönmez and Grote 2013). For software development projects, two fundamental types of uncertainty have been discussed:

requirements and technological uncertainty (Nidumolu 1995). Mellis et al. (2010) find that customers often have difficulties formulating specifications when following a sequential development process as many users and software developers do not have a clear understanding about all software details at the beginning of a project. Consequently, the team has to learn over time what to develop (Lyytinen and Rose 2006). Technological uncertainty is the second critical driver of software development complexity. As technology evolves over time and new technologies are being used, development teams often lack the necessary skills to work with new technology. However, the learning process is often unpredictable as developers have to learn while developing software (MacCormack and Verganti 2003).

Both, technological and requirements uncertainty are major influencing factors of software development projects complexity. To manage this complexity, software development teams either try to minimize project uncertainties by controlling impediments or to flexibly cope with them (Dönmez and Grote 2013). As the sources of uncertainty are often outside of control of the team, flexibility and the ability to adapt to new situations is an important determinant of successful software development teams. The software engineering discipline has proposed different *software development processes* to cope with uncertainty and project complexity. These processes specify general frameworks about structure and organization of software development projects.

2.1.2 Software Development Processes

Software development processes define “a set of activities that lead to the production of a software product” (Sommerville 2004, p. 64). As the development of different software systems may require different processes, the software engineering discipline has developed very different processes during the last decades. Nevertheless, there are several tasks that every development projects must include such as *definition tasks* for requirements specification, *implementation tasks* for software design, coding, and testing, and *evolution tasks* for modification, adaptations, and corrections. The different software development processes vary in terms of how strictly and in which sequences these tasks are addressed. Overall, three generations of software engineering development processes can be distinguished: craftsmanship, early software engineering, and modern software engineering.

Craftsmanship

Software development in the 1950s can be best described as *ad hoc* development with no standardized processes, technologies, or development methods. Products were customized for a particular purpose and deployed on mainframe computers. This led to various quality and maintenance issues (Austin and Devin 2009). Software development organizations used simple customized tools, processes, and

technologies to program machines with primitive languages. Later, the approach was an unformalized “code-and-fix” approach (Boehm 2006). While the first programmers mostly had an engineering background, more and more people from other disciplines started to develop software.

In the 1970s, the development process was formalized leading to the popular sequential “Waterfall” software development process (Royce 1970). It was a systematic engineering approach that adheres to specific process steps moving software through a series of representations from requirements to finished software (Boehm 2006). This approach assumes software development problems to be fully specifiable and optimal solutions to be predictable and planable in advance. Extensive plans were devised, processes were strictly followed in order to make development an efficient and predictable activity. The waterfall model was later interpreted and implemented as a sequential process with project-gates between clearly defined project phases. Design did not start before definition of a clear set of requirements and coding was not started before completion of the software design. The main idea was to shift from craft to industrial software production imitating manufacturing processes. Therefore, more and more components were built for reuse and process steps were standardized.

Early Software Engineering

In the 1980s, reuse of software functionality increased, new development tools were introduced, and new high-level object-oriented programming languages were developed improving developer productivity. In addition, software organizations used standardized processes to increase productivity. The industry had matured and was transforming into an engineering discipline.

In the 1990s, object-oriented methods were strengthened using design patterns. Modeling languages were introduced and quickly spread with the expansion of the Internet and the emergence of the World Wide Web. Organizations used more and more commercialized software such as operating systems, database systems, or graphical user interfaces and programmed most of the functionality in higher-level programming languages.

Modern Software Engineering

The importance of software as a discriminator of traditional products as well as internet-based software products required faster time-to-market times in the 2000s. In addition, the importance of user-interactive products made fast user feedback important which rendered the formal processes as too rigid. As a consequence, iterative development processes obtained more attention (MacCormack et al. 2001). Today, most of the software is built using standard tools and existing software functionality from commercial products or open source libraries. Typically, only

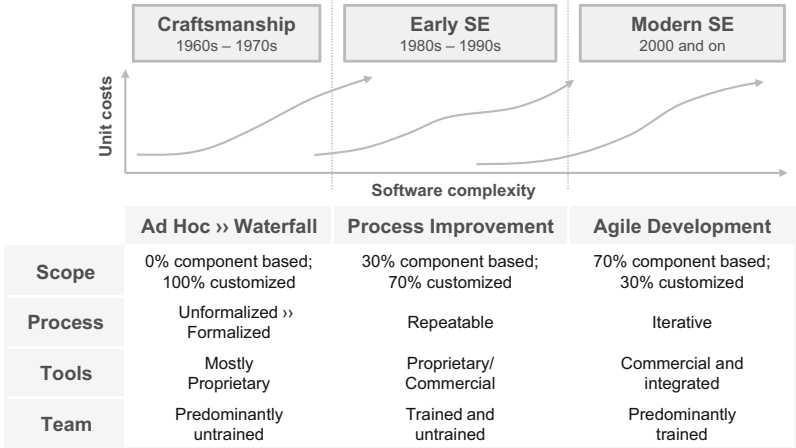


Fig. 2.1 Trends in software engineering (based on Royce et al. 2009)

about 30 % of the components need to be custom built (Royce et al. 2009) allowing more flexible processes.

Figure 2.1 illustrates the three generations of software engineering. The diagram on top of the figure schematically depicts the ability of each particular generation to handle software complexity in terms of cost per unit (Royce et al. 2009). As the complexity of software development projects has constantly risen over time, new approaches were introduced which better suited the given project contexts.

The software industry evolved from a “code-and-fix” paradigm to a professional industry. Many of these changes were incremental. But the late 1990s saw the emergence of a number of agile methods which meant a shift in software development processes and how many software development projects are organized today (Boehm 2006). This study is motivated by the agile software development paradigm. The underlying principles, agile methods, and agile practices are introduced in the following section.

2.2 Agile Software Development

The beginning of the 2000s saw a constant stream of change in the software industry. New technologies emerged and quickly adopted as a consequence of the exchange of ideas among developers through the global connectivity provided by the World Wide Web. The technological potential led to heavy investment into the IT industry. The software industry saw numerous mergers and acquisitions of existing as well as the rise of many new start-up companies. As a consequence, many software development projects faced tremendous organizational changes. In addition, more and more software applications were now developed for the

consumer market requiring user-friendly interfaces. For that, user feedback had to be quickly integrated into the development process leading to unpredictable and changing requirements. Overall, rapid change was becoming increasingly inherent to the software industry (MacCormack et al. 2001). Hence, speed-to-market and the ability to change to new requirements or react to customer feedback was essential to succeed in an environment of uncertainty (Baskerville et al. 2003). These challenges could only be met with shorter product life-cycles. As a consequence, the so-called lightweight methods evolved in the 1990s (Larman and Basili 2003) and represented an opposite pole to the heavy-weight plan-driven development processes which were soon considered as too rigid to successfully develop software in such volatile project conditions (Highsmith and Cockburn 2001)

2.2.1 Agile Values and Principles

In 2001, a group of 17 advocates of lightweight software engineering methods gathered to discuss their common grounds to coin the term “agile methods” in the so-called Agile Manifesto.⁵ It proposes a set of four core values for agile software development organizations. These agile values were derived from previous lightweight methods introduced by these agilists in the 1990s and early 2000s.⁶ The four values constitute the essence of agile software development:

Individuals and interactions over processes and tools⁷
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

Instead of formalizing the development process with detailed specification of software requirements, agile software development meant a distinct move towards continuous, informal, and close customer collaboration (Highsmith 2000). Unnecessary documentation was avoided as much as possible emphasizing a “lean” mentality adopted from lean manufacturing (Poppendieck and Poppendieck 2007). Agile developers would rather spend their time progressing the final software product instead of working on detailed project plans or extensive documentation of their software. Furthermore, people changed their perception seeing uncertainty

⁵See <http://agilemanifesto.org/>.

⁶Light-weight development methods proceeding agile software development: Adaptive Software Development (Highsmith 2000), Feature Driven Development (Palmer and Felsing 2002), Dynamic Systems Development Methods (Stapleton 1999), Scrum (Schwaber and Beedle 2002), Extreme Programming (XP) (Beck 2000), Lean Development (Poppendieck and Poppendieck 2007).

⁷Fowler (2002) explained that both sides of these statements are valued, but agile software engineering prefers the first over the second.

as an inherent part of software development as opposed to an unforeseeable contingency to be controlled through detailed upfront planning and compliance with strict processes (Dingsøyr et al. 2012).

2.2.2 Agile Methods and Practices

In the late 1990s and early 2000s, various software engineering methods were introduced. These methods are based on the idea of an incremental, iterative, and evolutionary software development process. As they encompass on the four aforementioned core values of agile software engineering, these methods were later called agile methods. Amongst them, *Scrum* and *Extreme Programming* are not only the most influential, but also the most popular today (Maruping et al. 2009a; VersionOne 2012).⁸ In the following, both are briefly introduced as they are at the core of this study.

Scrum

Schwaber and Sutherland (2011) define Scrum as a “framework within which people can address complex adaptive problems, while [...] delivering products”. Scrum is very popular amongst professional software development teams (VersionOne 2012). It is often referred to as a software development method, but strictly speaking, it is a project management framework. Scrum specifies (1) certain *roles* in the development team, establishes an (2) iterative work mode which centers around development *Sprints*, and defines different (3) *artifacts* that the developers use to coordinate their work. While first being published by Sutherland (1995) and later illustrated by Schwaber and Beedle (2002), the core concepts of Scrum are based on the ideas of Takeuchi and Nonaka (1986). All key elements of Scrum are illustrated in Fig. 2.2.

1. The project framework defines the *Scrum team* as a group of about ten people. There are two specific roles in the Scrum team: the Scrum Master and the Product Owner. The *Scrum Master (SM)* takes the role of a facilitator responsible to maintain the Scrum processes and eliminate impediments that might hinder the team from working efficiently. The *Product Owner (PO)* represent the customer within the team and voices customer requirements. Product owners define the team’s development targets in the coming Sprint and bear the responsibility to generate value for the customer. In their daily work, POs define customer

⁸Other agile methods are, for instance, Adaptive Software Development (Highsmith 2000), Feature Driven Development (Palmer and Felsing 2002), Crystal Clear (Cockburn 2005), or Kanban (Anderson 2004).

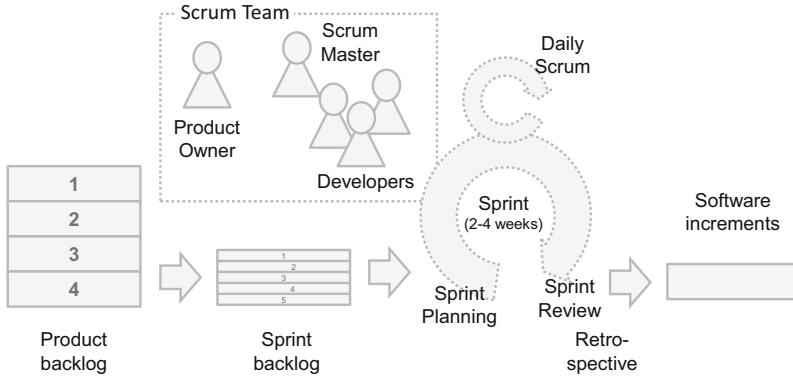


Fig. 2.2 Scrum development framework

requirements, define a list of prioritized development tasks for the team, and review work increments after each Sprint.

All other team members belong to the *development team* and do the actual software development work. They analyze requirements and design, develop, and validate the software. Scrum development teams are cross-functional, i.e. all team members are expected to have the necessary skill set to accomplish all of these software development tasks. Consequently, there are no additional roles in the development team such as user interface developers, testers, or other specialists.

2. Scrum teams pursue an iterative work mode and split the development project into short development cycles, so-called development *Sprints*. These Sprints have a specific length of 1–4 weeks, after which the team delivers new software features to the customer. This approach contrasts the traditional project with sequential phases for planning, development, validation, integration, and software release activities.

Every Sprint starts with a *Sprint planning* meeting during which the team decides on the features to be implemented. Subsequently, the team members specify sub-tasks and assign them to individual developers. All team members meet daily for about 15 min to synchronize their work and bring transparency to the work progress within the team. All developers inform the team about their accomplishments, describe the current work, and raise issues to be addressed by the team. Every Sprint ends with a *Sprint review* meeting during which the team presents its progress to the product owner or directly to the customer. In addition, the Scrum Master organizes a *Retrospective* meeting for the team to discuss possible improvement to teamwork processes in the future.

3. The team organizes its development tasks using a *product backlog*. The backlog contains a list of prioritized tasks defined by the product owner. The development team breaks this backlog into *sprint backlog items* and tracks its progress during each Sprint in a so-called *burndown chart*. This chart shows the ratio

of accomplished versus committed backlog items for that particular development Sprint.

Extreme Programming (XP)

XP is originally described by the authors as a lightweight method for small to medium-sized teams developing software in the face of vague or rapidly-changing requirements. Beck (2000) developed a set of programming practices while working on a project with Chrysler Group LLC in 1995. The key ideas are based on a set of values, principles, and practices developers should use to improve software quality and responsiveness to change. Developers constantly review system scenarios of the highest priority to business and quickly deliver the functionality (Fruhling and Vreede 2006). Amongst them, frequent releases of new software functionality to the customer and a constant focus on software quality is key. The general idea behind Extreme Programming is to take beneficial ideas and concepts of software engineering to “extreme” levels.

Extreme Programming received significant attention because of its emphasis on communication, simplicity, and testing, its sustainable developer-oriented practices, as well as its interesting name (Larman and Basili 2003). Extreme programmers advocate a strong focus on software code rather than plans or documentation. Furthermore, software quality is the main focus and the quality of the software should be permanently checked with automated tests. Unit tests check whether a particular piece of software works as intended, acceptance tests verify the satisfaction of user requirements, and integration tests validate coherent functionality of different modules of a software. Furthermore, extreme programmers keep the design simple and avoid overmanning features.

Extreme Programming proposes a set of software development practices. These include:

- **Pair programming**, two developers share a single workstation and collaboratively develop software side-by-side. One is actively writing code, the other observes, supports, and challenges the chosen approach in order to find better work results.
- **Code review**, the finalized software code is reviewed by at least one colleague prior to task finalization to obtain feedback and ensure high quality.
- **Test-driven development**, an iterative development practice where developers first write a test case for the wanted software functionality to verify if the software program includes the desired functionality. Only then, developers write software code to pass that test case.
- **Refactoring**, is a development practice including the restructuring of existing software code to improve its internal software quality, i.e. its readability or its structuredness, without changing its functionality. The objective is to increase the long-term maintainability and the extensibility of the software.

- **Continuous integration**, a software development practice where every developer working on a particular code base continuously integrates newly developed or changed software code to prevent integration problems. Mostly, integration tools are used that support the integration process.
- **Coding standards**, a set of rules or conventions of the developers in software development team or community. They include a common programming style which improve readability and the maintainability of a software code.
- **Collective code ownership**, a convention according to which everybody in a team or community owns the code, i.e. everybody is allowed to change any piece of code in a software. Simultaneously, every team member is responsible to ensure its quality.
- **Automated testing**, the use of special software to test the functionality of a software. Test cases check for expected outcomes of the tested software delivered given a set of inputs. These tests are executed for continuous feedback on the software functionality and different abstraction levels, for instance, unit, integration, or user interface tests.

2.3 Literature Review on Agile Software Development

Agile software development has been gaining popularity since the publication of the Agile Manifesto in 2001. Large software providers, such as Microsoft (Begel and Nagappan 2007), SAP (Schmidt et al. 2014; Schnitter and Mackert 2011), Adobe (Green 2011), and many others (VersionOne 2012) have adopted agile methods during the last years. As a consequence, agile software development can today be seen as a mainstream development methodology (West et al. 2010) with an increasing interest among professional software developers in rigorous validations of the effectiveness of the development approach.

During the last decade, researchers have been paying more and more attention to the phenomenon and studied diverse aspects of the agile software development paradigm. Most of these studies were either conducted by researchers in the software engineering (SE) or in the information systems development (ISD) community (Dingsøyr et al. 2012). Trends and findings of both research streams are described and discussed in the following paragraphs.

2.3.1 Information Systems Research

In contrast to the SE literature, there are no review articles about publications on agile software development in the Information Systems (IS) discipline. Therefore, a structured literature search was conducted to provide a comprehensive overview of existing publications. The results help to cluster main areas of interest and to analyze the research results. Beginning with the key words found in the software

Table 2.1 Number of articles found in the reviewed IS research outlets

Journals		Conferences	
European Journal of IS	10	American Conference on IS	14
Information Systems Journal	8	European Conference on IS	16
Information Systems Research	7	International Conference on IS	8
Journal of the Association for IS	1	Pacific Asia Conference on IS	3
Journal of Information Technology	1		
Journal of Management IS	3		
Journal of Strategic IS	0		
MIS Quarterly	2		
	Σ32		Σ40

engineering literature (see Table 2.5 on page 31), a list of search terms was defined:

- A {agile, agility}
- B {software, information system, information systems, IS}
- C {engineering, development}
- D {team, teams, method, methods, methodology, methodologies, project}

These search terms were combined as follows: {A1 OR A2} AND {B1 OR B2 ...} AND {C1 OR C2} AND {D1 OR D2 ...} to structurally search the top IS journals⁹ and conference proceedings for articles published between 2000¹⁰ and 2014. The resulting list of publications was complemented by articles found through a forward and backward search starting from the list of citations of the most relevant publications in the field. Appendix A.1 presents the full list of 72 papers which could be extracted from these outlets. A brief overview is provided in Table 2.1.

The increasing number of publications over the last years (see Fig. 2.3) shows a great interest in the topic amongst researchers in the international Information Systems community. Moreover, two special issues of the two leading IS journals within the last years (Abrahamsson et al. 2009; Ågerfalk et al. 2009) emphasize its importance (see Table 2.4). Nevertheless, the majority of publications on agile software development can be found in the software engineering outlets.

The extracted publications were analyzed for their research methodology, theoretical foundations, research context, and research focus. The findings are discussed subsequently.

Research Methodology Research on agile information systems development has a clear tendency towards qualitative research methods. About half of the reviewed studies were based on interview-based case studies. The conceptual papers (28 %)

⁹IS Basket of Eight: <http://aisnet.org/?SeniorScholarBasket>.

¹⁰The Agile Manifesto was published in 2001.

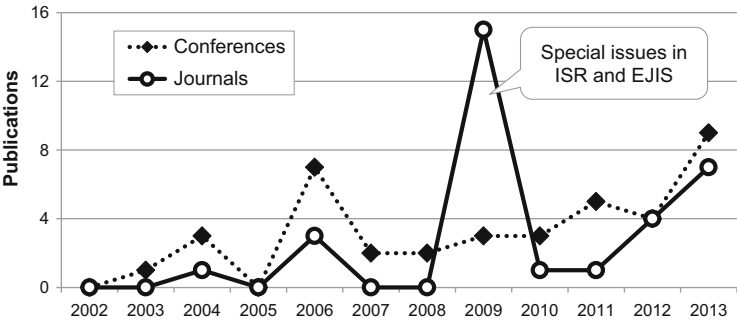


Fig. 2.3 Publications on agile ISD between 2002 and 2013 in the IS research community

are mainly concerned with the definition of agility or discuss the necessity for tools. Only 11 % of these papers report survey results (see e.g. Lee and Xia 2010; Maruping et al. 2009a). Overall, confirmatory research to test the explored results of the qualitative studies are mostly missing. The knowledge of the field is thus primarily derived from single teams working in very different project contexts.

Theoretical Foundations 44 out of 72 paper mention an underlying theory to guide the study. Overall, 29 different theories were found that were applied to understand the agile development approach. This demonstrates the clear response of the research community to the frequent calls for more theory-based studies on agile software development (Dingsøyr et al. 2012; Dybå and Dingsøyr 2008; Ågerfalk et al. 2009). The most frequently applied theoretical lenses were the *Theory of Complex Adaptive Systems* (Ralph and Narros 2013; Vidgen and Wang 2009; Wang and Conboy 2009) (7x), *Theory of Innovation Diffusion* (Mangalaraj et al. 2009; Schlauderer and Overhage 2013) (6x), *Control Theory* (Cram and Brohman 2013; Goh et al. 2013; Harris et al. 2009; Maruping et al. 2009a) (4x), and *Theory of Coordination* (Li and Maedche 2012; Strode et al. 2011)(3x). Furthermore, different theories and models from team effectiveness research have been used such as *Team Adaptation Theory* (Schmidt et al. 2013), *Group Think Theory* (McAvoy and Butler 2009), or *Leadership Theory* (Yang et al. 2009).

Research Contexts Most articles reported results from small, co-located teams, which develop new software products. Other studies researched agile software development aspects in distributed, large-scale, or maintenance project settings.

Research Foci The refined list of publications can be clustered into four main research topics (see Fig. 2.4). First, several articles were concerned with the conceptualization and definition of the software development agility concept. Second, other studies investigated the adoption and adaptation of agile methods in the field. Third, few studies examined the impact and role of standard project management topics, such as funding, leadership, or control of agile software development projects. Finally, different teamwork factors were examined in detail. Authors were interested

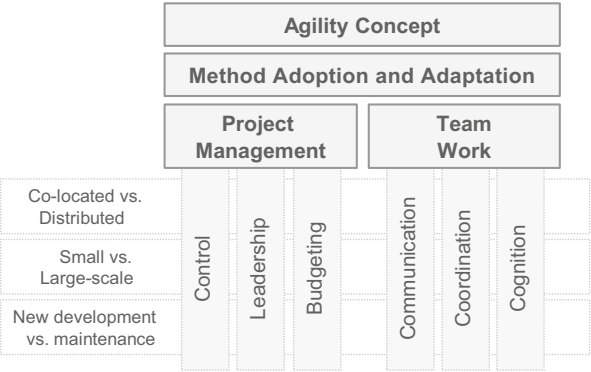


Fig. 2.4 Research foci on agile software development in the IS Community

in the influence of agile software development on specific teamwork factors and, in turn, the influence of these factors on the performance of agile teams. The following paragraphs provide an overview of the most influential publications for each category and summarize the main findings.

Agility Concept

Every good concept needs a strong underlying logic serving as a “theoretical glue” (Whetten 1989). In ISD research, however, it seems that “almost every piece of research adopts a unique interpretation of agility” (Abrahamsson et al. 2009). This conclusion was confirmed by Hummel (2014) who found 17 different definitions of the term after reviewing agile studies in the main outlets of software engineering in information systems research streams.

Among practitioners, the situation is not much different as the following developer quote, found in an internal collaboration forum of SAP SE, illustrates:

Agility is [in my company] the hyper buzzword these days. Everybody uses it, but nobody tells what is her/his understanding of it. I get goosebumps when I hear it. Some use it as a justification for a chaotic working mode. Some more for agile software engineering. Some see Scrum as part of it. Some not. Who knows. Actually I don't know what people - including [the Chief Technology Officer] - mean when they use that word.

Until 2009, little if any research has focused on the conceptual development of agility in ISD (Conboy 2009). One reason was that agile software development was motivated by the Agile Manifesto, which does not provide a clear definition of the concept. Instead, the four agile core principles are a non-formal definition of agility concept with general guidelines for software developers. In recent years, different perspectives on ISD agility have been provided by researchers without a compromise on definition and conceptualization of software development agility (van Oosterhout et al. 2006).

Table 2.2 Agile software development as a behavior

Abrahamsson et al. (2002)	Agility is defined in respect to the adoption of agile methods . Agile development methods are incremental, cooperative, straightforward, and adaptive
Conboy (2009)	Agility is defined in terms of the adoption of agile methods . Agility [of a method] is the continuous readiness to rapidly or inherently create change, proactively or reactively embrace change, and learn from change while contributing to perceived customer value
Maruping et al. (2009a)	“We use the term agile software development teams’ to refer to teams that are using an agile methodology ”
Qumer and Henderson-Sellers (2008)	“Agility is a persistent behavior or ability of an entity that exhibits flexibility to accommodate expected or unexpected changes rapidly; agility can be evaluated by flexibility, speed, leanness, learning, and responsiveness”
Schmidt et al. (2013)	A team’s behavior to iteratively and collaboratively accomplish its software development tasks including (1) software specification, (2) software design and implementation, (3) software validation, and (4) software release

The term agility was first used in the field of manufacturing research (Meredith and Francis 2000) and supply chain research. Within the IS discipline, it was first adopted by Overby et al. (2006) to describe the “sensing and responding capabilities” of a firm. Three years later, agility was adopted in the ISD context (Conboy 2009). Subsequently, several authors have provided their perspectives on software development agility since its initial introduction to the software development domain by the group of experienced practitioners in the Agile Manifesto. Researchers generally agree about the multi-dimensionality of the concept including a sensing and a responding dimension (Sarker and Sarker 2009). However, the meaning of agility itself in software development is yet to be fully understood (Börjesson and Mathiassen 2005). Three distinct conceptualizations of agility can be distinguished: Agility as a behavior, agility as a capability, and agility as an attitude.

Agility as a Behavior Several authors conceptualized the agility of a software development team by a team’s adoption intensity of agile development methods or development practices (see Table 2.2). Different approaches exist distinguishing agile and non-agile methods. Abrahamsson et al. (2002) suggested to describe incremental (small software releases), cooperative (close communication with the customer), straightforward (the methods are easy to learn and to modify), and adaptive (able to make last moment changes) software development methods as agile. Another, often-cited definition was developed by Conboy (2009), based on a thorough investigation of the agility concept in other research disciplines. Following his perspective, agility comprises two concepts, i.e. flexibility and leanness. Agility does not only incorporate the flexibility to change, but also the team’s ability to quickly respond to change. In addition, leanness is the contribution to perceived customer value through economy, quality, and simplicity. Taking this perspective,

software development teams are agile teams when adopting software development methods which lead to flexibility and leanness of the software development process.

Schmidt et al. (2013) proposed another perspective. They suggested to conceptualize agility of an software development team by its organization of central development task, such as specification, design, implementation, and software validation. Iterativeness and collaborativeness were suggested as the central behavioral markers of agile teams. Agile teams iterate the aforementioned tasks frequently while involving several team members in the process. In addition, agile teams plan, design, implement, and validate the software in small steps involving the entire team in all steps.

At a higher level of abstraction, Zheng et al. (2011) conceptualize agility as a “collective behavior, instantiated in improvisational behaviour of individuals and groups in their social interactions”. They further specify agility as “social actors [...] when engaging with uncertainty and complexity”.

Agility as a Capability The second perspective conceptualizes agility as a team capability (see Table 2.3). Agile teams are considered to possess the capability to effectively and efficiently react to change in the project context (Henderson-Sellers and Serour 2005) or, in a narrower sense, to react to changing customer requirements (Lee and Xia 2010). Other authors have specified particular team capabilities such as responsiveness, speed, competency, flexibility, and sustainability (Sharifi and Zhang 1999) or nimbleness, suppleness, quickness, dexterity, liveliness, or alertness

Table 2.3 Agile software development as a capability

Lee and Xia (2010)	“Software development agility is a team’s capability to efficiently and effectively respond to and incorporate user requirement changes during the project lifecycle”
Sarker and Sarker (2009)	“Agility in a distributed ISD setting is the capability of a distributed team to speedily accomplish ISD tasks and to adapt and reconfigure itself to changing conditions in a rapid manner”
Erickson et al. (2005)	“Agility is associated with such related concepts as nimbleness, suppleness, quickness, dexterity, liveliness, or alertness”
Henderson-Sellers and Serour (2005)	“Agility refers to readiness for action or change; it has two dimensions: (1) the ability to adapt to various changes and (2) the ability to fine-tune and re-engineer software development processes when needed”
Lyytinen and Rose (2006)	“Agility is defined as the ability to sense and respond swiftly to technical changes and new business opportunities; it is enacted by exploration-based learning and exploitation-based learning”
Vidgen and Wang (2009)	“Agile teams can be recognized by their ability to work with customers to coevolve business value, work sustainably with rhythm, be collectively mindful, create team learning, adapt and improve the development process, and to create product innovations”
Dingsør and Dybå (2012)	Team aspects of agility : “capability, talent, skill, and expertise to foster flexible anticipatory and reactive practices in response to changes in the state environment”

(Erickson et al. 2005) which can be assigned to agile teams. Lyytinen and Rose (2006) describe agility as an organizational capability to learn, explore, and exploit knowledge.

Sarker and Sarker (2009) combine a behavioral and ability perspective in their definition of agility for distributed ISD teams. On the one hand, they consider the capability of a distributed team to “speedily accomplish ISD tasks and to adapt and reconfigure itself to changing conditions in a rapid manner” as a key characteristic of agile teams. On the other hand, they define distinct behaviors of agile teams such as the right resources, the adoption of agile methods, and forging and maintaining of linkages across communicative and cultural barriers among distributed team members.

Agility as an Attitude Moreover, agility can be conceptualized in regards to the attitude of the members of software development teams. Change can either be perceived as a threat or an opportunity. Following this perspective, agile teams “embrace change as an opportunity and harness it for the organization’s competitive advantage” (Sharifi and Zhang 1999). Accordingly, agile teams expect and leverage change in the project context rather than assuming predictability. This perspective was not found in ISD literature, but in the manufacturing research stream only. Future research, however, might take this approach to study agile software development teams.

Given this conceptual diversity, Abrahamsson et al. (2009) concluded a need for every organization to appropriately define the agility concept specific to a given context. The same holds true for future research. A single interpretation may not be sufficient to advance research in the field. Nevertheless, a “solid platform on which to build a cohesive body of knowledge” is necessary for future cumulative research (Abrahamsson et al. 2009). This study follows the behavioral perspective and conceptualizes software development teams as agile when using agile development practices (see Sect. 4.3).

Agile Method Adoption and Adaptation

Several studies analyzed the adoption and adaptation of agile development methods and practices. Most studies assume that software development is restricted to small, co-located teams developing non-critical software. Few studies, however, investigate the reasons for developers’ deployment of agile methods in other contexts or their combination with the traditional, plan-driven development approach.

Method Adoption Mangalaraj et al. (2009) provide insights into individual, team, technological, task, and environmental factors that determine the acceptance of Extreme Programming practices. They identify individuals’ attitude and knowledge about the practices as essential determinants. In addition, development tools as well as the characteristics of the development task can support the adoption intensity. Finally, environmental factors such as time or budget constraints may influence how intensively the development practices are used. McAvoy and Butler (2009)

provide a decision support tool based on a “critical adoption factors matrix” to assess the suitability of agile methods in software development projects. The tool is based on insights from workshops with practitioners. Overhage and Schlauderer (2012) examine the long-term acceptance of Scrum, providing a list of acceptance factors evaluated in a single case study. Wang et al. (2012) conducted an exploratory study about the use of agile development practices. They take an ‘innovation assimilation perspective’ and describe four teams using agile methods during the acceptance, routinization, and infusion phases. Berger and Beynon-Davies (2009) investigate the usage of agile methods in large-scale software development teams. They assume the agile software development approach as primarily adopted by small development teams for short-term projects. In their case study, the authors demonstrate that agile software development can also be applied in the context of hierarchical decision-making, long-term projects, and high complexity. Austin and Devin (2009) provide a basic contingency framework based on the benefit/cost economics to discuss when to use an agile or plan-driven development approach. While most studies focus on new product development, Edberg et al. (2012) explore how information technology professionals define and select a methodology to maintain existing software using grounded theory. They provide a factor model to describe the decision process of software development teams between different components of standard methodologies. Lastly, Balijepally et al. (2009) focus on pair programming. In an experiment, they investigate reasons why and under which circumstances developers use pair programming. They find pair programmers to perform better than the second best programmer independent of the task complexity. Two developers, however, cannot exceed the performance of the best member working individually. In addition, the programmers show higher levels of confidence and satisfaction compared to the second best programmer, but not to the best programmer.

Method Adaptation Other studies examine the adaptation of agile methods to specific work contexts or their combination with the traditional, plan-driven software development approach. Cao et al. (2009) develop a framework for adapting agile development methods proposing a need for Extreme Programming practices to be adapted to different contexts. Based on adaptive structuration theory, the authors explain specific adaptations to address challenges of agile development teams. In a case study, Fitzgerald et al. (2006) exemplify this with Scrum and Extreme Programming at Intel. They propose these methods may complement each others’ incompleteness. Port and Bui (2009) run a simulation to better understand the benefits of combining the plan-driven and agile software development approaches for requirements prioritization. Mixed strategies, so they conclude, are likely to yield better results than pure agile or plan-based approaches. Karlsson and Ågerfalk (2009) discuss a formal and methodological approach to tailoring agile software development methods while emphasizing agile values and principles. Finally, Tanner and Wallace (2012) discuss how ISD teams adapt agile methods in distributed work contexts.

Agile Project Management

Different authors investigated project management factors in agile software development teams. These include control, leadership, and budgeting of agile teams.

Project Control Several authors apply control theory to study agile software development projects. Maruping et al. (2009a) examined the project conditions when agile methods are most helpful. Based on a survey with 110 software development teams, the authors find a beneficial use of agile methods in situation of frequent requirement changes and in control mode allowing teams to autonomously decide about the development activities. Goh et al. (2013) conducted a multiple case study. The publication includes a framework proposing project uncertainty and project urgency to be best addressed by an agile team using an interplay of team capabilities and trust-mediated organizational control mechanisms. Harris et al. (2009) studied when to provide teams with the flexibility to modify their directions and team external control of flexibility. The results demonstrate a need for flexibility in situations of uncertain starting conditions and the benefits of combining the traditional control mode with a newly proposed control mode to effectively manage such situations. Emergent outcome control was found to be specific to agile teams and different to behavioral and outcome control. It includes (1) scope boundaries constraining the solution space without specifying the solution and (2) ongoing feedback which is only provided when correction is needed. The studied agile methods were found to implement this control mode. Persson et al. (2012) studied the implementation of project control in a distributed software development team. Formal control practices were found to be enacted through communication media while clan control was predominantly exerted in informal roles and relationships. Cram and Brohman (2013) studied the influence of different development approaches on the control mode of software development projects. They provide a typology of ISD control modes differentiating between preventive and detective or corrective control practices, on the one hand, as well as product and process control objectives, on the other hand. The authors propose that agile projects tend to primarily utilize detective and corrective practices and combine them with process control mechanisms. A similar model was provided by Gregory et al. (2013).

Project Leadership Yang et al. (2009) compare leadership in agile and traditional software development teams to find managers of agile teams in higher need of a transformational leadership style to achieve success. Transformational leaders develop their followers focusing on motivation, morale, and job performance. Transactional leaders, on the other hand, provide rewards for accomplishments and make sure followers comply with defined standards.

Project Budgeting Cao et al. (2013) emphasize that a “just enough planning” mentality in agile project may aggravate funding decisions of project managers. The authors present a framework to explain how organizations’ adaptation to the funding approach to accommodate the characteristics of agile projects. According to

their results, funding decisions should be based on continuous feedback from project team members and negotiations based on changing customer values. These decision may be implemented through contracts with fixed prices or negotiated scope or pay-as-you go models. Keaveney and Conboy (2006) propose a cost estimate model for agile projects. Essential for the success is expert knowledge and analogy to past projects. Moreover, the authors find fixed price budgets in agile projects to be beneficial for developers and customers.

Agile Teamwork

Most information systems are too large to be developed by a single person. Hence, several developers collaborate in a single or multiple software development teams. During the last 60 years, team effectiveness research has provided extensive knowledge about work teams. The research resulted in many different teamwork models and theories about work team effectiveness (Cohen and Bailey 1997). In recent years, several researchers have built on this knowledge by opening the black-box of agile software development teams and examining the effect on different teamwork aspects. The most popular topics are communication, coordination, and cognition in agile teams. Most of these studies were published in conference outlets indicating opportunities to further development of theses studies. To date, several studies are still at a research-in-progress stage or first steps to an evolving research field.

Team Communication Rosenkranz et al. (2013) assume the ability to communicate and to reach a shared understanding between the software customer or users and developers at the heard of requirements development. The authors propose the quality of language as a suitable means for “the emergence of coherent and meaningful requirements”. The authors provide research propositions and suggest to analyze language use and communication in requirements development in future studies. Hummel and Rosenkranz (2013) propose “social agile practices” to positively influence the communication behavior of an ISD team which, in turn, may lead to higher mutual understanding and better relationship in the team leading to project success. The authors develop a set of research proposition and provided a measurement model for empirical tests, left for future research.

Team Coordination Xu and Cao (2006) investigate coordination mechanisms in agile software development teams. They distinguish between vertical and horizontal coordination mechanisms, both proposed to determine the performance of agile teams. Vertical coordination involves formal coordination by supervisors while horizontal coordination involves peer-to-peer coordination between team members. Strode et al. (2011) define the concept of “coordination effectiveness” in agile development teams as the “state of coordination wherein the entire agile software development team has a comprehensive understanding of the project goal, the project priorities, what is going on and when, what they as individuals need to do and when, who is doing what, and how each individuals work fits in with other team

members work”. Li and Maedche (2012) build on this idea and study coordination effectiveness in distributed agile development projects.

Team Cognition Different studies focused on knowledge distribution and knowledge sharing in software development teams. Maruping et al. (2009b) conducted a survey study with more than 500 developers working in 56 teams and found that the two agile practices collective ownership and coding standards moderate the relationship between expertise coordination and software project technical quality. Moreover, collective ownership thereby attenuates the relationship and coding standards strengthen the relationship. The underlying theoretical propositions of the study were derive from transactive memory systems literature. Other studies found “collective mindfulness” to be important in agile ISD teams. According to McAvoy et al. (2013), mindfulness promotes a focus on “continuous attention to detail” and “vigilance to minimize errors and respond effectively to unexpected events”. As a consequence, the new perspective allows to study agility in terms of “being agile” rather than “doing agile”. Finally, Spohrer et al. (2013) study the influence of pair programming and peer code review on the creation of knowledge in software development teams.

2.3.2 Software Engineering Research

The interest of the software engineering research community in agile software development is primarily evident from the growing number of scientific publications since 2001. Figure 2.5 illustrates the number of publications either published in conference proceedings or scientific journals (Dingsøy et al. 2012). The most popular conference outlets are the International Conference on Agile Software Development (“XP”), the International Conference of Product Focused Software Development and Process Improvement (“PROFES”), and the International Confer-

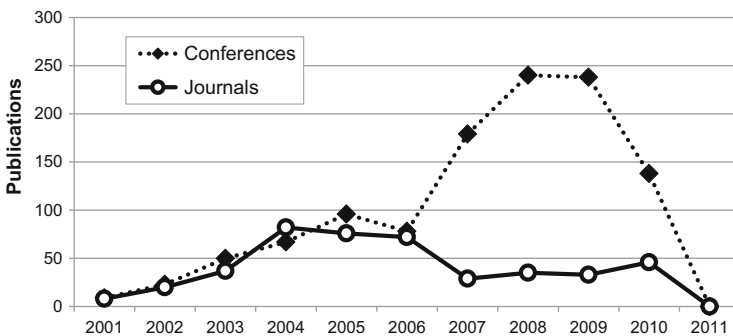


Fig. 2.5 Publications on agile software development between 2001 and 2010 in the SE research community (source Dingsøy et al. 2012)

Table 2.4 Special issues on agile software development in IS and SE outlets

Outlet	Introductory article	Field
IEEE Computer	Williams and Cockburn (2003)	SE
Journal of Database Management	Siau (2005)	SE
Software Practices and Experience	Greer and Hamon (2011)	SE
Journal of Information and Software Technology	Dybå (2011)	SE
Journal of Systems and Software	Dingsøyr et al. (2012)	SE
Information Systems Research	Ågerfalk et al. (2009)	IS
European Journal of IS	Abrahamsson et al. (2009)	IS

SE - software engineering outlets, *IS* - information systems outlets

ence on Software Engineering (“ICSE”). The major scientific journals publishing on agile software development include IEEE Software, Journal of Systems and Software, Information and Systems Technology, and Empirical Software Engineering. In addition, several special issues addressed the topic indicating a keen interest in the field of software engineering (see Table 2.4).

The findings of these articles were comprehensively summarized by various review articles during the last decade. Table 2.5 on page 31 provides an overview of these review articles and briefly summarizes their conclusions.

Authors from several countries and diverse background have contributed scientific studies for a better understanding of agile software development (Chuang et al. 2014). Moreover, several special issues in leading SE journals have addressed the topic and there are periodical conferences on the topic, e.g. International Conference on Agile Software Development. While the first studies were mainly best-practice reports or success stories (Abrahamsson et al. 2002), more and more studies are published with scientific rigor and relevance (Sfetsos and Stamelos 2010). Nonetheless, “most [of the studies] are inspired by practices emerging in industry” (Dingsøyr et al. 2012).

Chuang et al. (2014) classified about 70 % of the studies as case studies, design simulations, or experiments. Of note, many of these case studies were either conducted with student teams or with new teams; only few studies investigated mature teams (Dybå and Dingsøyr 2008). The advantage of case study research is an analysis of a specific phenomenon at the potential cost of generalizability of the results.

The first studies addressed questions about the adoption of agile methods. In a second phase, the impact on the software development process was investigated. Some studies found positive effects on team learning, knowledge exchange, and improved software code quality due to the use of agile practices such as pair programming. However, pair programming was also found to be “extremely inefficient”, “very exhausting”, and “a waste of time” (Tessem 2003). Various studies found effects on better collaboration in the team, an improved “faith in their own abilities”, respect and trust in software development teams (Robinson and Sharp 2005). Some studies investigate the role of personality of software developers

Table 2.5 Publications on agile software development in SE outlets

Review	Period	Scope	Findings/Conclusions
Abrahamsson et al. (2002)	≤ 2002	Experience reports	<ul style="list-style-type: none"> – Only a limited number of empirically validated studies found – Studies are mainly with anecdotal evidence from “success stories” – Agile methods are “effective and suitable for many situations” – More empirical studies to evaluate effectiveness needed
Cohen et al. (2004)	≤ 2004	Experience reports, research studies	<ul style="list-style-type: none"> – Insights from workshops, experiments, surveys, and seven case studies – Adoption and first experience with Extreme Programming and Scrum – Agile will not replace the traditional development approach – Both paradigms will find their areas of applications
Erickson et al. (2005)	≤ 2005	Experience report, research studies	<ul style="list-style-type: none"> – Mainly case studies about Extreme Programming – Most case studies promote the success of Extreme Programming – Need for more structured research in comparison to other approaches – “hard, empirically-based economic evidence is lacking”
Dybå and Dingsøyr (2008)	≤ 2005	Research studies	<ul style="list-style-type: none"> – Thirty three studies found with acceptable rigor – Four streams: <ul style="list-style-type: none"> (a) introduction and adoption, (b) human and social factors, (c) customer and developer perception, (d) comparative studies – Focus on human and social factors suggested for future studies – Lack of theoretical and methodological rigor – Need for a research agenda to increase quality and quantity of studies
Sfetsos and Stamelos (2010)	≤ 2009	Research studies	<ul style="list-style-type: none"> – Forty six empirical studies with “acceptable rigor, credibility, and relevance” – Three streams: studies on (a) test driven development, (b) pair programming, (c) XP practices – Test automation: better software quality as key benefit – Pair programming: better code quality and improved teamwork

(continued)

Table 2.5 (continued)

Review	Period	Scope	Findings/Conclusions
Jalali and Wohlin (2012)	1999–2009	Experience report, research studies	<ul style="list-style-type: none"> – Studies in distributed settings with globally distributed teams – Reviewed studies are mostly industry experience reports – Most studies focus on a particular Extreme Programming practice – Comprehensive framework to understand agile SE is needed
Dingsøyr et al. (2012)	2001–2010	Research studies	<ul style="list-style-type: none"> – Research community shows growing interest in agile SE – Limited number of studies with theoretical support – (a) knowledge management, (b) personality theories, or (c) learning theories – However: “general perception that agile research tends to be a-theoretical” – “Urge to embrace a more theory-based approach in the future when inquiring agile development”
Chuang et al. (2014)	2001–2012	Research studies	<ul style="list-style-type: none"> – Review of the key outlets and contributions to agile ISD literature – About half of the studies are case studies; only 7 % are surveys – Only 2 % pursue theory development – “Research on agile development methods remains at the infancy stage” – Call for more industrial and scholarly research studies
Hummel et al. (2013b)	≤ 2013	Research studies (IS & SE)	<ul style="list-style-type: none"> – Review about the role of communication in agile IS development – Communication process within agile ISD is still not well understood – Theories of communication, collaboration, cognition, and sense-making for future research

(Young et al. 2005). Other studies investigated developers’ perception and found 95 % of the employees using XP would like their company to continue using agile methods (Mannaro et al. 2004). Finally, several studies compare the productivity of the agile versus the traditional software development showing positive to neutral impact (Ilieva et al. 2004; Wellington et al. 2005) while most studies found a positive impact on software quality (Layman et al. 2004; Wellington et al. 2005).

Critical studies on agile software development question the novelty of agile software development, criticize a lack of focus on long-term architecture, claim that it would only fit to small software development teams, and predict that Extreme Programming may lead to inefficient teamwork (Dybå and Dingsøyr 2008; McAvoy and Butler 2009).

Most of the studies in the software engineering outlets are neither concerned with theory development nor do they refer to existing theories from other fields to explain the effectiveness of agile software development. Instead, they are mainly descriptive with a lack of theoretical and conceptual foundation in the research stream (Abrahamsson et al. 2009). Only recently, few articles were published in a special issue of the *Journal of Systems and Software* exploring the “theoretical underpinnings of agile development” (Dingsøyr et al. 2012). The focus was on coordination, decision making, and social factors in agile software development. In conclusion, four key results can be found in the software engineering literature:

- The software engineering research community shows great interest in the topic
- Studies are mostly case studies
- Studies are mainly descriptive with no or limited theory-based explanations
- Many studies find evidence for a positive impact on software quality and teamwork factors, but there is a lack of integrated or cumulative research

2.4 Discussion of the Literature

In recent years, many software development organizations have shifted their development processes to agile software development (VersionOne 2012). As a consequence, agile software development can be described as mainstream today (West et al. 2010). At the same time, research on agile software development has matured from “practitioners’ success-stories” (Abrahamsson et al. 2002) to a covered domain in the software engineering as well as the information systems research community. Both communities have shown particular interest in the topic as the extensive list of publications (see Figs. 2.3 and 2.5) and various special issues on agile software development indicate (see Table 2.4).

In 2008, Dybå and Dingsøyr (2008) reviewed the research body of knowledge on agile software development. They found a number of research studies, but also identified the need for more rigorous studies to advance the evolving field. In the meantime, several researchers have responded to this call resulting in an increasing number of publications on agile software development. Many of these studies are exploratory in nature and draw their insights from single or multiple case studies (see Appendix A.1). In addition, most of them explore small, co-located software development teams or projects while only few studies researched distributed or large-scale development settings. The exploratory character of the field allows researchers to better understand the implementation of agile software development. Against this background, this study aims to provide generalizable research findings to advance the research field.

Agile software development is a phenomenon primarily driven by experienced practitioners and consultants in the software industry (Boehm 2006). Hence, several researchers have first discussed how and why software development organizations use and adapt agile methods (see section “Agile Method Adoption and Adaptation”).

Moreover, they evaluated how customers', developers', or students' perception of the agile development approach. Other studies assessed the development approach in terms of productivity, software quality, and job satisfaction (Dybå and Dingsøyr 2008). Many of these studies were primarily descriptive in nature.

Until today, the research community is still far away from fully understanding why, how, or in which project contexts agile software development works. In other words, there is still a long way for researchers to develop a theoretical understanding of agile software development. Such a theoretical perspective could not only explain the success of agile software development, but also guide professionals when and how to use the agile development approach.

A structured search of studies in the IS field revealed that more than 60 % of the publications build on a theoretical model or framework (see Appendix A.1). This shows that more and more studies seek to contribute to theory development in this field of research. This study intends to contribute to this literature stream and advance the theory-based understanding of agile software development.

As with every evolving research field, key concepts need to be discussed to ensure a common understanding in the research community for cumulative research success. The last years have seen valuable contributions to this discussion (see section "Agility Concept"). Various perspectives have been provided and there is agreement of agility being a multifaceted construct (Sarker and Sarker 2009). However, there is no agreement what agility is or how it should be conceptualized. This is one reasons why research on agile software development is still fragmented, as the boundaries and key concepts are not aligned. Another reason may be a lack of a clear definition of the dependent variable of many studies. While some studies are interested in the impact on software quality, delivered scope, in-time or in-budget delivery, other are curious to understand the impact on communication, coordination, or knowledge structures in software development projects or teams.

Overall, researchers have studied very diverse aspects of agile software development. Missing boundaries as well as a missing understanding about the definition of software development agility complicates the integration of the research stream. Therefore, this study intends to clearly define how it conceptualizes the multifaceted concept of agile software development as well as how it defines software development team performance (see Sect. 3.3.1).

In conclusion, agile software development is a software development approach with growing popularity since the early 2000s. Research on agile software development has advanced from best-practice success stories to more rigorous research studies with a predominantly descriptive character. The extant knowledge of teamwork research may provide a promising theoretical lens for this research direction. Due to the relevance for software development organizations, such a model should not only be generalizable but also validated with data from professional software developers. Since research on agile software development is very fragmented, every study needs to clearly define the conceptualization of agility and the dependent variable. Only then, follow-up studies can build on the study's results to advance the field towards an integrated research prospect. Dingsør and Dybå (2012) summarized the field of research in a call for future studies. They demanded (a) a better

measurement of teamwork aspects in software development, (b) more rigorous industrial cases, (c) better understanding of dynamic configurations, (d) increased emphasis on team cognition, and (e) a better understanding of multicultural context of software development teams to advance the understanding of agile software development teams. This study seeks to contribute to the reviewed literature streams addresses these research challenges.

The author agrees to these conclusions seeing both an interest among researchers and a need for further research on agile software development. First, the results of the extensive literature review indicate an imbalance of applied research methods skewed towards qualitative research with only a limited number of quantitative studies. Second, theory-guided research on agile software development remains limited and many studies are still based on experience lacking theoretical support. Despite the strong focus on teamwork and collaboration in agile software development teams, only few studies draw on the extensive body of knowledge about the effectiveness of work teams to better understand agile software development. Finally, there is still no agreement among researchers about the definition of agility. As a consequence, study results are not comparable leading to a lack of cumulative studies and knowledge about agile software development. Overall, there is a need for more rigorous, theory-supported studies with insights from professional software developers.

Agile Software Development Teams

Schmidt, C.

2016, XIV, 184 p. 67 illus., 35 illus. in color., Hardcover

ISBN: 978-3-319-26055-6