2012 45th Hawaii International Conference on System Sciences

Identifying Common Characteristics in Fundamental, Integrated, and Agile Software Development Methodologies

Sebastian Dyck
Accenture

sebastian.dyck@accenture.com

Tim A. Majchrzak
Department of Information Systems
University of Münster
tima@ercis.de

Abstract

Many software development projects fail to reach their goals or are aborted. This results in economic problems, lost welfare, and may even endanger humans. In many cases, an improper choice of the software development methodology (SDM)—or not following an ideology towards development at all—can be identified as the root problem. Choosing an adequate methodology is no trivial attempt, though. Conflicting aims of classical methodologies and agile approaches add further complexity. We identify common characteristics that can be found in each SDM and present a structured approach for classification. We also present a mapping to existing methodologies to underline the applicability of our approach. This helps to compare fundamental, integrated, and agile SDMs. Furthermore, we discuss our findings and highlight their implications for choosing an SDM. We thereby contribute to the theoretical body of knowledge and provide advice for businesses at the same time.

1. Introduction

Failing software development projects are a common observation. Even though only some failures are noticed by the general public—such as the modernization of US taxing authorities with a delay of eight years and a loss of (at least) 1.6 billion USD [1]—studies of software development failure fill up books [2], [3], [4]. Failure can be observed independently of project kind and size, and industrial sector [5]. This not only is an economical problem for affected companies and a loss in welfare but in some cases humans are harmed directly. A well studied example is the introduction of a new system for ambulance dispatching in London. Delayed dispatching due to system problems has been attributed for up to 30 fatalities [6], [7]. The observed problems were *systematic* rather than coincidental [8].

There are two ways to address such problems. Firstly, *Software Engineering* (SE) applies principles of the engineering discipline to software development [9]. This specifically concerns the architectural and technical background of development. Examples for SE practices are the use of design patterns [10] and the introduction of software testing techniques [11]. Secondly, software development methodologies (SDM) provide frameworks for managing development projects. They do not primarily address technical details of how software is developed but determine the organizational embedding of development and give guidelines for organizing activities. Thereby, they can be used to decrease the risk of project failure [12].

Since Benington proposed the *Phase Model* in 1956 [13], a great number of development methodologies have been made up [14], [15]. It is indisputable that no silver bullet [16] has been found so far—development methodologies should be chosen (and customized) in accordance with projects' characteristics [17], [18]. Applicability to problem domains is often associated with agile development [19]. Discussions—particularly whether to develop agile, or not—tend to be dogmatic [20], though, and no structured approach for choosing development methodologies exists [21]. The latter has been identified as a research objective [22].

Our work contributes to this research in several ways. Firstly, we present the theoretical underpinning of software development methodologies. We explicate the characteristics that can be used to describe methodologies and provide a framework for categorization. Secondly, we provide an exemplary mapping to existing methodologies. This also highlights differences and similarities between fundamental, integrated, and agile approaches. Thirdly, we discuss our findings to highlight implications and directions for future research. And fourthly, we contextualize the topic by embedding it into a survey of existing (yet limited) approaches.

The paper is structured as follows. Section 2 summarizes existing work. Our research method is sketched in Section 3. In Section 4 we develop common characteristics for software development methodologies and map them to actual SDMs. Section 5 presents a discussion of our findings and highlights future work. In Section 6 we draw a conclusion.

2. Background and Existing Work

For illustration purposes, we relate our argumentation to several widely used software development methodologies.

978-0-7695-4525-7/12 $26.00 © 2012 IEEE
DOI 10.1109/HICSS.2012.310

These are the *Waterfall Model* according to Royce [23] and Boehm [24], the *Spiral Model* according to Boehm [25], [26], the *Rational Unified Process* (RUP) according to Booch et al. [27], [28], the V-Model XT [29], *Scrum* according to Schwaber [30], [31], *Extreme Programming* (XP) according to Beck [32], [33], Crystal according to Cockburn [18], [34], Feature Driven Development according to Palmer and Felsing [35], and Dynamic Systems Development Method of the DSDM Consortium [36]. We expect the reader to by roughly familiar with these SDMs. Empirical validation for the prevalence of the chosen methodologies is e.g. given by Cockburn [34].

First attempts to identify criteria for choosing SDMs can be traced back to the 1980s [25], [37]. An early concept was proposed by Boehm and Belz in 1990 [38]. They present a model that is based on five steps and takes into consideration a number of criteria. While this work was ahead of his time, it is limited to the by-then popular methodologies such as the Waterfall Model and the Spiral Model.

Alexander and Davis presented an approach for choosing adequate processes for software development projects in 1990 [39]. By limiting their work on processes, not all aspects of SDMs are addressed. Nevertheless, 20 criteria from different categories were identified. For each criterion three possible values can be chosen. Turner and Boehm [40] [41] proposed a set of selection criteria for SDMs in 2003. Their work is driven by the insight that software development is highly influenced by human factors such as culture and communication. They also stress the influence of human factors on their choice of agile methodologies.

Filß et al. describe 18 criteria to support decisions for or against development methodologies [42]. They also propose a process for identifying SDMs that might be suitable for a company. The work can be used for classification but it lacks details on the actual choice of SDMs. In their 2008 work, Bunse and von Knethen suggest how to chose SDMs [43]. Their approach is based on first determining the business context and to then mapping it with methodologies. This mapping is based on 11 criteria. The selection takes into account qualitative data only and is not entirely transparent.

Wildemann classifies nine SDMs by proposing 12 criteria with five distinct values each [21]. Unfortunately, it is not described how values have been determined. Development projects are classified by 33 criteria. Choosing a SDM is based on a comparison of the methodologies' applicability.

Finally, Heinemann and Engels [44] describe a methods for project-specific selection of a SDM. Considered methodologies include the Waterfall Model, the Rational Unified Process, and Scrum. The method is based on a relatively small number of criteria. Nevertheless, the authors report empirical evaluation and underline its practicability.

The history of classification approaches underlines the importance of the topic. At the same time, it shows that no unified approach has been found, that no holistic method for selection has yet been developed, and that most ideas either lack a sound theory or empirical validation.

Besides the structured approaches, research on differences of methodologies and their applicability to certain kinds of projects exist. For example, a not scientifically assessed study of *NTT data* compares three different development methods [45]. Three teams of nine developers each had to develop a program. The study particularly reports a shorter time for finishing the program by developing *agile*. However, the given case can be explained by the *Follow-the-sun* approach [46], [47] chosen by the agile team; the team that followed the waterfall model only worked 8 hours per day. Since this example and similar work only roughly relate to our paper, we will not highlight further approaches.

3. Methodological Approach

The research method our work is based on is *design science* [48]. While empirical studies make an important contribution to the knowledge on SDMs, our work has a different aim. By identifying common characteristics—which can also be called IT *artifacts* [48] in this context—, we foster a general understanding of software development methodologies. We followed the typical cycle of *build* and *evaluate* [49]. Characteristics were identified iteratively by analyzing the literature and checking existing methodologies. We then evaluated our findings by applying them to actual SDMs, eventually leading to improvements of the artifacts [50]. This process in reflected in Section 4.

In the overall context of our research, this paper is a first step. The ultimate goal is to design a method that allows to identify the most suitable SDM in a given context. This includes to be able to categorize applied methodologies by merely checking their characteristics. In particular, it should become possible to understand how companies modify methodologies for their needs and to learn whether these modifications have a satisfying impact. Reaching this goal requires a number of subsequent steps, each contributing to the body of knowledge and broadening the understanding of SDMs. Therefore, the iterative nature and the problem solving focus of design science [49] is ideal for our work.

A particularity of our work is that its outcome was only clear to some extend a priori. An approach that is based on "learning via making" [51] is typical for design science research. A final reason for using a design-oriented approach is the creation of artifacts "intended to solve organizational problems" [52]. Our work also contributes to the theoretical knowledge of IT artifacts—namely the characteristics and SDMs. Theorizing about IT artifacts has been identified as a vital function of information systems research [53]. At the same time, the result of our work is valuable for businesses.

The previously presented existing approaches on the selection of development methodologies utilize different levels of abstraction. The approach in [40] mainly focuses on

balancing of traditional and agile methodologies, whereas the approaches in [38], [39], [43], [21] and [44] operate on the level of concrete methodologies. Similar to [42], in this paper we lower the level of abstraction and directly focus the characteristics of methodologies in an unaggregated way. This approach increases the transparency as it reveals the root cause for the suitability of a specific method in an actual project. Moreover, this level of detail enables the users of a certain methodology to identify inappropriate characteristics as well as possibilities for tailoring and customization. The way characteristics were derived is not described from an high level perspective. Rather, we explain it along with each characteristic in the following section.

4. Software Development Methodologies

After explaining basic ideas, we explain characteristics of SDMs. They are then exemplarily mapped to existing SDMs.

4.1. Basic Considerations

By definition, every software project is characterized by its uniqueness described by its objectives and scope, its specific constraints on time, budget and resources as well as its designated form of organization. However, one can identify uniform and recurring patterns across projects. Thus, software development methodologies represent a supportive and structuring element for running of software projects.

Therefore, we define a *software development methodology* (SDM) as a reference model for the development of software describing the various statuses of the corresponding software projects. According to Shuja and Krebs, methodologies typically include *process elements* such as phases, iterations, milestones, and activities as well as *content elements* such as work products, roles, standards, and practices [54]. Based thereupon, we identified different attributes and corresponding values. Taking on a *process-oriented* perspective, the identified attributes include *supportive disciplines*, *development phases*, and a *proceeding strategy*.

In contrast, taking on a *content-oriented* perspective, the identified attributes are *specification depth*, *mandatory practices*, and *procurance assistance*.

In the style of the classification of Ramsin and Paige, we distinguish between

- fundamental methodologies[1] such as the Waterfall Model and the Spiral Model,
- integrated methodologies such as the Rational Unified Process and the V-Model XT,
- agile methodologies [55] such as Scrum, Feature Driven Development, Extreme Programming, Dynamic Systems Development Method, and Crystal Clear.

1. Ramsin and Paige call them *seminal* [55] which reflects their importance but not their character.

This distinction resembles the historical development of SDMs from very basic approaches to integrated ones and the rather new emergence of agile methodologies.

4.2. Characteristics of Development Methodologies

In the following sections we described the distinctive characteristics methodologies are made of.

4.2.1. Supportive Disciplines. The examined methodologies primary focus on aspects of software development such as analysis, implementation, and testing. However, various methodologies additionally describe supportive and thus related disciplines such as project management, quality management, change management, configuration management or documentation.

Although there are general frameworks for the supportive disciplines such as the *Project Management Body of Knowledge* (PMBoK) or *Projects in Controlled Environments* (PRINCE2) for project management and *ISO 9001* and *Software Process Improvement and Capability Determination* (SPICE) for quality management, SDMs cover those subjects in an integrated and dedicated manner.

4.2.2. Development Phases. In order to structure software projects, factually related activities are usually grouped into different *phases* that represent the life cycle of a software system—ranging from its initial idea via its realization and operations to its final disassembly. In this context, the IEEE defines the *software development cycle* as an aggregation of the phases *requirements*, *design*, *implementation*, *testing*, and *installation* [56]. Building on that, the concept of the *software life cycle* additionally covers the post-assembly phases *operations*, *maintenance*, and *disassembly*.

However, software development methodologies do not necessarily cover all of these phases. This especially holds for the additionally defined phases of the *software life cycle*. In fact, commonly known methodologies such as the Rational Unified Process and Scrum indeed cover all development-related phases but they provide marginal information on the subsequent ones. However, decoupling phases of development from phases of operations, maintenance, and disassembly may result in critical risks and issues such as

- inefficient collaboration between development and operations teams,
- insufficient maintenance-related work products in case of production problems, and
- deprecated documentation and legacy source code hindering disassembly and replacement.

In order to overcome these drawbacks, the following three approaches can be identified.

1) *Tight Coupling* with related concepts aiming at easy integration. For instance, the *V-Model XT* [29] explicitly defines an interface for post-development phases

conducted with the *Information Technology Infrastructure Library* (ITIL) [57].

2) *Defined Extension* of an existing software development methodology. For instance, the *Enterprise Unified Process (EUP)* by Ambler and Constantine extends the Rational Unified Process with information on operations and maintenance phases [58].
3) *Pragmatic Combination* of a software development methodology with an analogue concept. For instance, the utilization of Scrum with *Kanban* can be frequently observed as they both represent lightweight and agile approaches [59].

Thus, coverage of phases as well as the capability for coupling, extension, and combination with other concepts represent a differentiation criterion among software development methodologies.

4.2.3. Proceeding Strategy. Bremer classifies software development methodologies via their *proceeding strategy* [60]. In this context, the author identifies the values *sequential*, *iterative*, *incremental*, *participatory*, and *evolutionary*:

- *Sequential.* The phases of the software development methodology are executed one after another. In order to start a certain phase, its precedent phase must have been completely finished first. It is generally distinguished between a strictly sequential strategy and a sequential strategy with offsets. The latter explicitly allows to repeat previously already finished phases.
- *Iterative.* In contrast, iterative software development methodologies cyclically conduct and finish the defined phases. The main objective of methodologies with this particular proceeding strategy is to receive early customer feedback in order to identify shortcomings as soon as possible. Figure 1 exemplarily shows that all components are already implemented after the first iteration. The later iterations only serve for *refinements*.

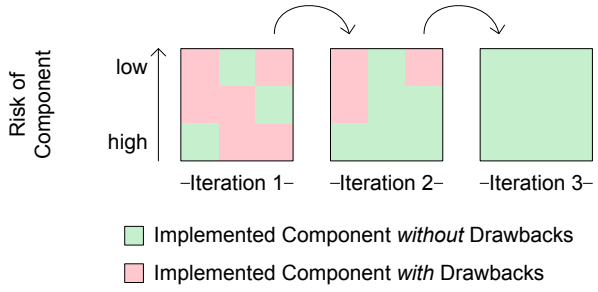

Figure 1. Iterative proceeding strategy

- *Incremental.* Analogously to the iterative proceeding strategy, the phases are cyclically conducted. However, after each cycle only a particular increment is delivered. Therefore, these increments respectively represent a subset of the whole system. In order to continuously minimize the risk of the project, the most critical components are usually implemented in the early iterations. Figure 2 exemplarily shows how increments primary serve as *extensions*, not as refinements.

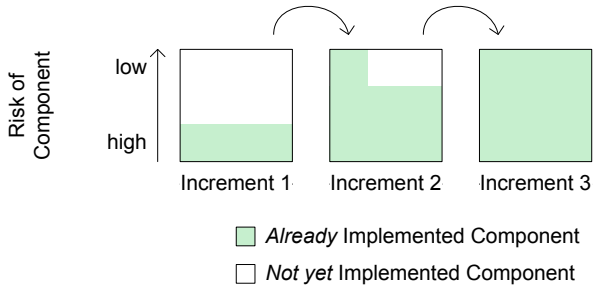

Figure 2. Incremental proceeding strategy

- *Participatory.* The prospective users of the system are actively and continuously involved during the whole development process. Critical success factors for this proceeding strategy are users access to relevant information, the possibility of taking independent positions, involvement in decision making, adequate methods for participatory development, and sufficient tolerance regarding technical and organizational solutions.
- *Evolutionary.* Considering this particular proceeding strategy, the initially not entirely known requirements are successively identified via practical tryout. In contrast to the iterative and incremental proceeding strategies, the objectives of the project are not fully established initially but identified and refined throughout the development process. An evolutionary proceeding strategy may be realized using *evolutionary prototyping*.

Different combinations of these proceeding strategies can be observed in theory and practice—the values of attributes are not disjunctive. For instance, the Rational Unified Process is classified as iterative *and* incremental. Figure 3 exemplarily shows an iterative-incremental proceeding strategy.

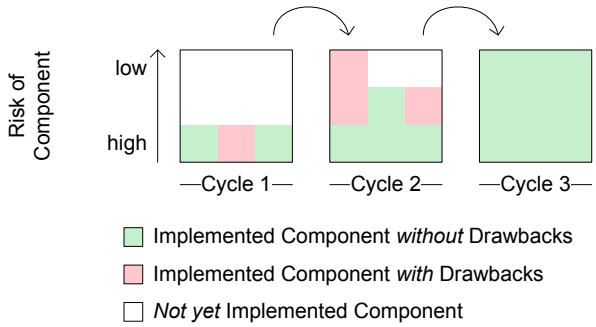

Figure 3. Iterative-incremental proceeding strategy

However, the emergence of SDMs with a proceeding strategy that is both iterative and incremental led to some confusion on their differences so that even synonymous usage of these two terms can be observed. Even though both strategies conduct the project phases in cycles, they differ in their main objectives—iterations represent *refinements* whereas increments represent *extensions*.

4.2.4. Specification Depth. The present controversy on fundamental, integrated, and agile software development methodologies is also a debate on the depth of the solution specification in order to successfully conduct a software project. Considering this attribute, Bunse and von Knethen identify the disjunctive values of learning teams, defined efficiency rules, predefined starting suggestions, and formalized frameworks [43]. In order to further differentiate methodologies belonging to the latter class, we additionally distinguish between those formalized frameworks that have an abstract character and those with a very detailed embodiment. Figure 4 graphically summarizes this definition; the values can be described as follows:

- *Learning Teams.* Representatives of this value have the highest degree of flexibility as there are only very few defaults specified. Instead, the project teams are asked to conjointly and adaptively organize their work in an efficient, target-oriented way. An example of a learning teams-centered methodology is Scrum.
- *Defined Efficiency Rules.* This approach follows close rules aiming at the efficient conduction of a software project. Rules are dependent on defined influence factors such as the size of a team, and scale from a strong emphasis on flexibility to a more detailed focus *(Build-up)*. The Crystal methodology family [61] is considered as a methodology with defined efficiency rules.
- *Predefined Starting Suggestions.* Methodologies with this specification depth support project managers and teams by the means of predefined starting suggestions. Suggestions include related requirements that represent the basis for all involved stakeholders for the individual embodiment of the actions within the software project. Extreme Programming is an exemplary methodology with predefined starting suggestions.
- *Abstract Frameworks.* Methodologies representing abstract frameworks are characterized by a large set of defined core elements, roles, activities, and work products. However, abstract frameworks provide no or only little information on the actual embodiment as regards content. Therefore, these frameworks offer a certain degree of freedom of design. The V-Model XT can be classified as an abstract framework.
- *Detailed Frameworks.* The highest degree of detail is covered by representatives of this class. Methodologies with a specification depth like this provide a broad portfolio with defaults and descriptions on various aspects of the project. They also cover supplementary elements such as metrics, standards, and checklists. It is generally required to tailor (i.e. *customize*) detailed frameworks by selecting all required elements in a project-specific manner *(Cut-down)*. The Rational Unified Process can be considered to be a detailed framework.

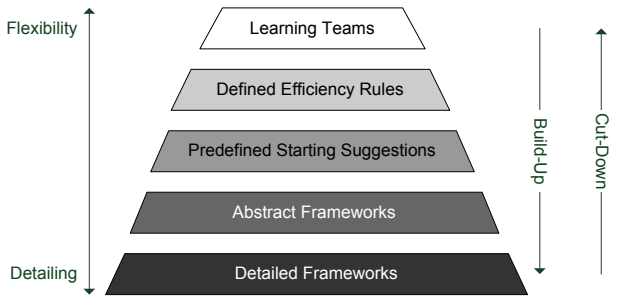

Figure 4. Different specification depths

4.2.5. Mandatory Practices. Practices, sometimes differentiated as *best practices* and *good practices*, represent specific instructions on various activities in software projects. Such practices often address only particular aspects of a wider problem. For instance, coding standards may enable standardization and simplification of the source code whereas test-first programming can represent an instrument for the reduction of occurring defects. However, both practices can separately contribute to an adequate inner and outer software product quality. Following the approach of Jacobson et al. [62], we distinguish the following classes of practices:

- *Software Engineering Practices.* These practices aim at the analysis, design, and implementation of software artifacts. Examples are *Visual Modeling* in the Rational Unified Process and *Prototyping* in the Spiral Model.
- *Social Engineering Practices.* Communication, coordination, and collaboration of project teams is the aim of these practices. Examples are *Stand Up-Meetings* in Scrum and *Side-by-Side Programming* in Crystal Clear.
- *Organizational Practices.* These practices focus on planning, executing, and controlling software projects. Examples are *Timeboxing* in the Dynamic Systems Development Method [36] and *Negotiated Scope Contracts* in Extreme Programming.

Due to its' concretely described activities and objectives, practices generally have a very application-oriented nature. Therefore, this characteristic considerably separates practices from more abstract elements of SDMs such as values and principles. Moreover, in most cases practices are independent of other practices for which reason they can be applied or not applied based on the concrete situational context of a project. Nevertheless, in this paper we primary focus on defined *mandatory practices* of software development methodologies in order to sharpen our research's focus.

Even though the relevance of practices increased particularly due to the emergence of agile software development methodologies, many ideas realized in practices can be traced back to fundamental and integrated approaches. Therefore, some sources estimate that there are several hundred practices in existence [63]. However, as practices enjoyed their renaissance due to the emergence of agile SDMs, it can be differentiated between process-oriented agile methodologies such as Scrum, practice-oriented methodologies such as Extreme Programming as well as hybrid ones putting emphasis on both processes, and practices as it is the case in Feature Driven Development.

4.2.6. Procurance Assistance. In order to successfully introduce and utilize a software development methodology, the provided capabilities for the procurance assistance need to be considered. Hence, the different project stakeholders need to acquire role-specific knowledge on the corresponding methodology. In this facet of organizational learning, we distinguish *mediated* and *applied* procurance assistance.

The available media on software development methodologies ranges from summarizing articles via comprehensive books to detailed handbooks. Moreover, electronic process guides (EPG) such as the Eclipse Process Framework (EPF) [64] have been established. In contrast to traditional offline documentation, these tools enable methodologists to first tailor and then publish a methodology in a digital manner. Afterwards, the resulting information can be consumed from different perspectives, for instance in the form of activities and roles structured along roles or phases.

Applied procurance assistance can be mainly subsumed as the availability of training, coaching, and certifications. In this context, trainings can be characterized as an proactive type of knowledge transfer describing the desired *to-be* situation. In contrast, coaching primary features the observation and analysis of the present *as-is* situation and the subsequent identification of areas for improvement. Moreover, certifications are considered as an instrument for the demonstration of the capabilities of individuals and organizations towards a specific software development methodology.

4.3. Exemplary Mapping to Existing Development Methodologies

In order to illustrate our classification framework, we map the previously described attributes and values to selected popular representatives of each class of software development methodologies, namely

- the *Waterfall Model* as a traditional methodology,
- the *Rational Unified Process* as an integrated methodology, and
- *Scrum* as well as *Extreme Programming* as agile methodologies.

We selected two representatives of agile methodologies in order to illustrate that there may be fundamental differences even in methodologies of the same class that can be outlined using the identified characterics and values.

4.3.1. Waterfall Model. Taking on a process-centered perspective, the supportive disciplines of the Waterfall Model are mainly quality management and documentation. In this context, some sources criticize that testing activities start too late and without any customer involvement [65]. However, a central concept of the Waterfall Model is the integrated verification and validation of the results by the customer in order to complete a certain phase [23]. An underlying implication of the Waterfall Model is that both problem and solution can be fully described upfront [66], i.e. a priori. This is also articulated by the emphasis of the Waterfall Model on extensive documentation [23]. Considering other process-oriented characteristics, the Waterfall Model describes a *sequential* proceeding strategy also covering all phases from requirements to operations, while not explicitly covering maintenance and disassembly [23].

From a content-oriented perspective, we classify the specification depth of the Waterfall Model as *abstract framework*. The methodology strictly defines the different, sequentially ordered development phases, various work products and five principles such as *design-first development* and *customer involvement* [23]. However, it does not provide in-depth information on how to practically realize these guidelines. Additionally, only the usage of the practice of prototyping is considered as mandatory practice. Procurance assistance is only available in the form of articles such as [23] and [24] or some summarizing book chapters.

4.3.2. Rational Unified Process. The Rational Unified Process is structured in core phases covering all activities from initial business modeling and requirements engineering up to the final deployment. However, additional information on operations, maintenance and disassembly is not fully included. Considering the supportive disciplines of Rational Unified Process, the methodology also explicitly defines supplementary phases covering the supportive disciplines *project management*, *change management*, and *configuration management* [27]. Moreover, the methodology also considers various activities for quality management and documentation in an integrative manner [42]. As the Rational Unified Process is an indirect descendant of the spiral model [67], it also follows an *iterative*, *incremental*, and *evolutionary* proceeding strategy.

In its most recent version 7.0, the Rational Unified Process describes about 35 roles, 55 activities, and 70 work products as well as additional assistance, for instance in the form of guidelines and templates [28]. In comparison with other methodologies, the description of these defined elements is considered as very comprehensive [41]. As one of the

key principles of the RUP is its cut-down project-specific tailoring approach, the specification depth is classified as *detailed framework*. Additionally, the methodology defines a couple of practices, among them *prototyping*, *visual modeling*, and *self-organizing teams*. Considering the procurance assistance, the guidelines for the Rational Unified Process is available as books such as [27] and [68], handbooks such as [28], and as the electronic process guide *Rational Method Composer* [69]. Additionally, trainings, coachings, and certifications are offered.

4.3.3. Scrum. Scrum has a strong emphasis on project management [41]. Similar to other agile methodologies, Scrum is designed to be responsive to changing requirements and thus provides assistance for change management. However, other supportive disciplines like quality management, configuration management and documentation are not fully covered by this methodology [70]. Scrum does only briefly describe the different software development life cycle phases. The proceeding strategy of Scrum is defined via so called *sprints*, aiming at an *iterative*, *incremental*, and *evolutionary* delivery of the final software product.

From a content-oriented perspective, Scrum can be classified as a methodology of *learning teams*. As it only describes very few roles, activities, and work products, it strictly follows a build-up approach. Moreover, Scrum only describes a few social engineering and organizational practices such as self-organizing teams, daily stand up meetings and frequent delivery. However, it does not define any software engineering practices [41]. Scrum is published in the form of articles and books such as [30], [31] and [71]. The *Eclipse Process Framework* [64], an electronic process guide, can be extended with a freely available Scrum library. Additionally, various opportunities for trainings, coachings, and certifications are offered.

4.3.4. Extreme Programming. Extreme Programming is generally considered as a very developer-centric methodology. The supportive disciples of project management and documentation are not fully covered by Extreme Programming [72]. However, the methodology defines various principles and practices for the partial coverage of quality management, change management, and configuration management. Moreover, Extreme Programming only describes the development phases *requirements*, *design*, *implementation*, and *test*. Similar to Scrum, the philosophy of Extreme Programming is based on the assumption that it is generally not possible to fully identify and describe all requirements of a software project a-priori [33]. Therefore, the methodology is based on an iterative, incremental, and evolutionary proceeding strategy. Consequently, Extreme Programming explicitly emphasizes the intensive involvement of the customer. Therefore, Extreme Programming requires the continuous availability of authorized customer representatives [33]. In the first version of Extreme Programming, it was even demanded to always have a dedicated customer representative on-site [32]. Thus, the proceeding strategy is also considered as *participatory*.

The content of Extreme Programming is based on the five values *simplicity*, *communication*, *feedback*, *courage*, and *respect*. Derived from these, it additionally defines 14 principles. In order to be easily applicable, these principles are realized as 24 different practices such as prototyping, test-first programming, frequent delivery and pair programming. Besides that, it only defines very few roles, activities, and work products leading to a specification depth that can be classified as *defined starting suggestions* [43]. The primary source for Extreme Programming is represented by the books [32] and [33] and credited to Kent Beck. Furthermore, there is an early version of the Extreme Programming library for the Eclipse Process Framework available. Additionally, various trainings and coachings exist. However, no certification program has been set up, yet [33].

5. Discussion, Prospects, and Future Work

Our work is not finished with identifying characteristics and mapping them to existing SDMs. In fact, it leaves much room for discussion—and it is a foundation for future work.

Following the literature and particular practitioners' discussions on methodologies, idealogical bias is a common observation. Classical (sometimes called *traditional*) SDMs and agile approaches seemingly contradict in any possible dimension (cf. e.g. [73], [74]). A shift towards agile methodologies even seems to have an *emotional* component [75]. Our findings acknowledge that there are significant differences between fundamental, integrated, and agile methodologies. Nevertheless, they also show that characteristics can be shared between classical and agile approaches. As suggested by Jiang and Eberlein, there is a *relationship* between them [76] that goes beyond contradicting ideas. Moreover, there can be mixed approaches. Evidence for this claim has already been reported by other researchers [77].

Hardly any model will be implemented by practitioners in a way that completely resembles the theoretical description of it. Due to the nature of companies, business, and software engineering, there always will be nuances—which can be better understood with the work provided in this paper. Additionally, companies may choose to implement a SDM but also follow some ideas from other methodologies. It would be possible to encounter implementations of classical SDMs in combination with the adoption of agile ideas. For example, there is no inherent reason to refrain from using *pair programming*, a typical technique of Extreme Programming [33], in combination with classical SDMs.

Whereas the identification of common characteristics is finished, our work has raised a number of questions. As sketched earlier, the characteristics can be used to design a

method for identifying feasible SDMs based on the situation met in a company and based on a development project's characteristics. For instance, the following characteristics can be considered:

- project-related characteristics such as project size, problem complexity, project priorities (optimization towards functionality, quality, cost, or duration), project criticality (ranging from loss of comfort up to loss of lives), dependencies to other projects and project transition, responsibility for operations, and maintenance (project team or dedicated line organization),
- people-related characteristics such as team size, team experience on existing methodologies, functional team experience in the problem domain, technical team skill in the solution domain, and geographic distribution (on-site or off-site customer or team), or
- environment-related characteristics such as the customer's cooperate culture (plan-driven, agile), customer empowerment, completeness of requirements, and expected level of change requests.

The method requires empirical validation both regarding its usability and its helpfulness for companies. For this goal, the design science approach that is suitable for building and continuously refining a framework to chose SDMs will have to be accompanied by qualitative and quantitative research.

The embodiment of our framework can be demonstrated using the example of the correlation between the usage of the pair programming practice in combination with the project complexity, project priorities, and the technical team skill in the solution domain. In their meta-analysis on the effectiveness of pair programming, Hannay et al. interpreted various experiments on the usage of this practice [78]. On an aggregated level, the authors identify a positive overall effect of pair programming on quality, a negative overall effect on effort and a positive overall effect on duration. However, the authors also point out that these effects of pair programming increase in correlation with increasing problem complexity and decreasing developer skill.

Applying these findings on our framework, we can identify that pair programming fits well for projects with a high problem complexity, a priority of optimizing towards quality or duration and medium or low skill of the technical team in the solution domain. Hence, software development methodologies defining pair programming as a mandatory practice would be considered as generally more suitable for a software project with these particular characteristics. However, in order to identify the most suitable methodology as a starting basis for further project-specific tailoring, also the correlation between other characteristics of methodologies with characteristics of projects need to be taking into account. Subsequently, these results can be aggregated for decision making, for instance using a *scoring model*.

Following Turner's and Boehm's idea of homegrounds for the various software development methodologies [41], we outline the following exemplary hypotheses for the identified and outlined characteristics:

- A software development methodology with an explicit coverage of the change management supportive disciplines is best applicable for a software project with a large project size, high dependency on other projects, a low customer empowerment, low completeness of requirements, and a high expected level of change requests.
- A software development methodology with a tight coupling approach for the integration with operations and maintenance phases is best applicable for projects with a project priority on quality, a dedicated line organization responsible for operations and maintenance, and a plan-driven customer's cooperate culture.
- A software development methodology with a participatory proceeding strategy is best applicable for projects with a high problem complexity, low functional team experience in the problem domain, an on-site customer availability, agile customer cooperate culture, high customer empowerment, low completeness of requirements, and a high level of expected change requests.
- A software development methodology with a specification depth of self-organizing teams is best applicable for software projects with low project size, low complexity, low criticality, no dependencies to other projects, low team size, high team experience on existing methodologies, no geographic distribution, and an agile customer cooperate culture.
- A software development methodology offering an electronic process guide for procurance assistance is best applicable for software projects with a high project size, a high team size, a low team experience on existing methodologies, and a plan-driven cooperate culture.

In addition to the presented work, the identified characteristics could be augmented with more specialized ones. For example, using development tools or the focus on industrial sectors might have impact on the choice of SDMs. Similarly, the mapping of characteristics to the company context can be refined—and refinements should then be evaluated. Eventually, a tool could be developed that allows project managers to specify a project's background and goals. It would recommend suitable methodologies and explain their impact and the estimated effort for adopting them.

Our future work will address some of the mentioned questions. At the same time, we hope to encourage other researchers to contribute to this field of research. Particular for the empirical evaluation of the findings and for studies about the adoption of SDMs by companies, the work of teams from various countries and with different background would be highly beneficial. We specifically deem our work useful for the agile community and for researchers that compare agile approach to classical ones (cf. e.g. [79]).

6. Conclusion

We presented work on the characterization of software development methodologies. Based on a thorough analysis of the background and of related work, we introduced our design science approach. We iteratively identified common characteristics of SDMs and presented them as a framework. Each methodology can be checked for its core ideas with regard to a number of characteristics, namely for supportive disciplines, development phases, proceeding strategy, specification depth, mandatory practices, and procurance assistance.

In order to both illustrate and evaluate our approach, we applied it to actual methodologies and highlighted the possible mapping. The mapping to the Waterfall Model, the Rational Unified Process, Scrum, and Extreme Programming demonstrate the feasibility of our work. Besides, these sections provide many sources for further reading. Finally, we discussed the impact of our findings.

A number of questions has been raised and future work in various directions can be based on our work. Particularly the development of a method to allow companies to select project-specific SDMs that serve their needs regarding project duration, budget, and quality is very appealing—yet highly non-trivial. Our framework can also be combined with other authors' work or used in conjunction with it. A combination of theoretical approaches and empirical work is specifically attractive. Our work will continue with refinements and extensions of our framework. Moreover, we will try to empirically validate our findings and to apply them to settings in companies.

References

[1] R. L. Glass, *Software Runaways: Lessons Learned from Massive Software Project Failures*. Upper Saddle River, NJ, USA: Prentice Hall, 1998.

[2] R. L. Glass, *Computingfailure.com: War Stories from the Electronic Revolution*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001.

[3] R. L. Glass, *In the Beginning: Recollections of Software Pioneers*. Los Alamitos, CA, USA: IEEE CS, 1997.

[4] R. L. Glass, *Computing calamities: lessons learned from products, projects, and companies that failed*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1999.

[5] S. Flowers, *Software failure: management failure: amazing stories and cautionary tales*. New York: Wiley, 1996.

[6] D. Page, P. Williams, and D. Boyd, *Report of the Inquiry into the London Ambulance Service*. South West Thames Regional Health Authority, 1993.

[7] P. Beynon-Davies, "Information systems 'failure': the case of the London Ambulance Service's Computer Aided Despatch project," *EJIS*, vol. 4, no. 3, pp. 171–184, 1995.

[8] A. Finkelstein and J. Dowell, "A Comedy of Errors: the London Ambulance Service case study," in *Proc. 8th Int. WS on SW Spec. and Design*. IEEE CS, 1996, pp. 2–4.

[9] P. Naur and B. Randell, *Software Engineering: Report of a Conference spon. by the NATO Science Committee, Garmisch, Germany*. Scientific Affairs Division, NATO, 1969.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns. Elements of Reusable Object-Oriented Software*. München: Addision-Wesley, 1995.

[11] M. Pezze and M. Young, *Software Testing and Analysis: Process, Principles and Techniques*. Wiley, April 2007.

[12] P. Beynon-Davies and M. D. Williams, "The diffusion of information systems development methods," *The Journal of Strategic Information Systems*, vol. 12, no. 1, pp. 29–46, 2003.

[13] H. D. Benington, "Production of large computer programs," *IEEE Ann. Hist. Comput.*, vol. 5, pp. 350–361, October 1983.

[14] R. Ramsin, *The Engineering of an Object-Oriented Software Development Methodology*. York, UK: University of York, 2006.

[15] J. Rittinghouse, *Managing Software Deliverables: A Software Development Management Methodology*. Newton, MA, USA: Digital Press, 2003.

[16] F. P. Brooks, Jr., *The mythical man-month (anniversary ed.)*. Boston, MA, USA: Addison-Wesley, 1995.

[17] J. Charvat, *Project Management Methodologies: Selecting, Implementing, and Supporting Methodologies and Processes for Projects*. Wiley, 2003.

[18] A. Cockburn, *Agile Software Development: The Cooperative Game*, 2nd ed. Addison-Wesley, 2006.

[19] J. Highsmith, *Agile software development ecosystems*. Boston, MA, USA: Addison-Wesley, 2002.

[20] J. Coldeway, "Beratung: Die allein selig machende lehre," *ObjektSpektrum*, no. 02/2001, pp. 86–88, 2011.

[21] H. Wildemann, *Prozessgestaltung in der Softwareentwicklung: Leitfaden und Tools zur effizienten Entwicklungsprozessgestaltung*, 9th ed. München, Germany: TCW, 2008.

[22] R. L. Glass, "Matching methodology to problem domain," *Commun. ACM*, vol. 47, pp. 19–21, May 2004.

[23] W. W. Royce, "The development of large software systems," in *Proc. IEEE WESCON 1970*. IEEE CS, 1970, pp. 328–338.

[24] B. W. Boehm, "Software engineering," *IEEE Transactions on Computers*, vol. 25, no. 12, pp. 1226–1241, 1976.

[25] B. W. Boehm, "A spiral model of software development and enhancement," *Software Engineering Notes*, vol. 11, no. 4, pp. 14–24, 1986.

[26] B. W. Boehm, "A spiral model of software development and enhancement," *Computer*, vol. 21, no. 5, pp. 61–72, 1988.

[27] P. Kruchten, *The Rational Unifed Process: An Introduction*, 3rd ed. Addison-Wesley, 2003.

[28] A. K. Shuja and J. Krebs, *IBM Rational Unified Process Reference and Certification Guide*. IBM Press, 2008.

[29] R. Höhn and S. Höppner, Eds., *Das V-Modell XT: Grundlagen, Methodik und Anwendungen*. Berlin: Springer, 2008.

[30] K. Schwaber, "Scrum development process," in *Proc. 10th OOPSLA*. ACM, 1995, pp. 117–134.

[31] K. Schwaber, *Agile Project Management with Scrum*. Microsoft Press, 2004.

[32] K. Beck, *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[33] K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change*, 2nd ed. Addison-Wesley, 2004.

[34] A. Cockburn, *Crystal Clear: A Human-Powered Methodology for Small Teams*. Addison-Wesley, 2005.

[35] S. R. Palmer and J. M. Felsing, *A Practical Guide to Feature-Driven Development*. Prentice Hall, 2002.

[36] F. P. Miller, A. F. Vandome, and J. McBrewster, *Dynamic Systems Development Method*. Alpha Press, 2009.

[37] V. R. Basili and H. D. Rombach, "Tailoring the software process to project goals and environments," in *Proc. of the 9th Int. Conf. on Software Engineering*, 1987, pp. 345–357.

[38] B. W. Boehm and F. Belz, "Experiences with the spiral model as a process model generator," in *Proc. 5th Int. ISPW on Exp. with Software Process Models*. IEEE CS, 1990, pp. 43–45.

[39] L. C. Alexander and A. M. Davis, "Criteria for selecting software process models," in *Proc. 15th COMPSAC*, 1991, pp. 521–528.

[40] R. Turner and B. W. Boehm, "People factors in software management: Lessons from comparing agile and plan-driven methods," *CrossTalk The Journal of Defense Software Engineering*, pp. 4–8, 2003.

[41] B. W. Boehm and R. Turner, *Balancing agility and discipline: a guide for the perplexed*. Addison-Wesley, 2003.

[42] C. Filß, R. Höhn, S. Höppner, M. Schumacher, and H. Wetzel, "Rahmen zur auswahl von vorgehensmodellen," in *Entscheidungsfall Vorgehensmodelle*, R. Petrasch, R. Höhn, S. Höppner, H. Wetzei, and M. Wiemers, Eds. Aachen: Shaker, 2005, pp. 183–227.

[43] C. Bunse and A. von Knethen, *Vorgehensmodelle kompakt*. Spektrum, 2008.

[44] M. Heinemann and G. Engels, "Auswahl projektspezifischer vorgehensstrategien," in *17. Workshop der Fachgruppe WI-VM der GI*, O. Linssen, T. Greb, M. Kuhrmann, D. Lange, and R. Höhn, Eds. Aachen: Shaker, 2010, pp. 132–142.

[45] S. Ueberhorst, "Eine Aufgabe – drei Entwicklerteams," *Computerwoche*, vol. 45, no. 10, 2011.

[46] E. Carmel, *Global software teams: collaborating across borders and time zones*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1999.

[47] E. Carmel, Y. Dubinsky, and A. Espinosa, "Follow the sun software development: New perspectives, conceptual foundation, and exploratory field study," in *Proc. of the 42nd HICSS*. Washington, DC, USA: IEEE CS, 2009, pp. 1–9.

[48] H. A. Simon, *The sciences of the artificial*, 3rd ed. Cambridge: MIT Press, 1996.

[49] A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design Science in Information Systems Research." *MIS Quarterly*, vol. 28, no. 1, 2004.

[50] A. Hevner and S. Chatterjee, *Design Research in Information Systems: Theory and Practice*. Springer, 2010.

[51] B. J. Oates, *Researching Information Systems and Computing*. Sage Publications, 2005.

[52] B. Niehaves and B. Stahl, "Criticality, Epistemology, and Behaviour vs. Design – Information Systems Research Across Different Sets of Paradigms," in *14th European Conference on Information Systems (ECIS 2006)*, Gothberg, Sweden, 2006.

[53] W. J. Orlikowski and C. S. Iacono, "Research Commentary: Desperately Seeking the "IT" in IT Research–A Call to Theorizing the IT Artifact," *Information Systems Research*, vol. 12, no. 2, pp. 121–134, 2001.

[54] A. K. Shuja and J. Krebs, *IBM Rational Unified Process Reference and Certification Guide*. IBM Press, 2008.

[55] R. Ramsin and R. F. Paige, "Process-centered review of object oriented software development methodologies," *ACM Computing Surveys*, vol. 40, pp. 1–89, 2008.

[56] IEEE, The Institute of Electrical and Electronics Engineers, Inc., "IEEE Std 610.12-1990: IEEE Standard Glossary of Software Engineering Terminology," New York, 1990.

[57] n. A., *The Official Introduction to the ITIL 3 Service Lifecycle: Office of Government Commerce*. The Stationery Office, 2007.

[58] S. W. Ambler and L. L. Constantine, *The Unified Process Transition and Production Phase: Best Practices in Implementing the UP*. CMP Books, 2002.

[59] C. Ladas, *Scrumban – Essays on Kanban Systems for Lean Software Development*. USA: Modus Cooperandi, 2009.

[60] G. Bremer, "Genealogie von Entwicklungsschemata," in *Vorgehensmodelle für die betriebliche Anwendungsentwicklung*, R. Kneuper, G. Müller-Luschnat, and A. Oberweis, Eds. B. G. Teubner, 1998, pp. 32–59.

[61] A. Cockburn, *Crystal Clear: a human-powered methodology for small teams*. Addison-Wesley, 2005.

[62] I. Jacobson, P. W. Ng, and I. Spence, "Enough of processes – lets do practices," *Journal of Object Technology*, vol. 6, pp. 41–66, 2007.

[63] R. Hauber, I. Jacobson, S. Mühlbauer, and I. S. Pan-Wei Ng, "Practices als Alternative zu vollständigen Prozessen," *ObjektSpektrum*, vol. 03/2009, pp. 62–69, 2009.

[64] Eclipse Foundation, "Eclipse Process Framework Project (EPF)," 2011, online: http://www.eclipse.org/epf/.

[65] B. Hindel, K. Hörmann, M. Müller, and J. Schmied, *Basiswissen Software-Projektmanagement*, 3rd ed. dpunkt, 2009.

[66] W. W. Agresti, "The conventional software life-cycle model: Its evolution and assumptions," in *New Paradigms for Software Development*. IEEE CS, 1986, pp. 2–5.

[67] B. Boehm, "A view of 20th and 21st century software engineering," in *Proc. ICSE '06*. New York, NY, USA: ACM, 2006, pp. 12–29.

[68] P. Kroll and P. Kruchten, *The rational unified process made easy: a practitioner's guide to the RUP*. Addison-Wesley, 2003.

[69] IBM, "Rational Method Composer," 2011, online: http://www-142.ibm.com/software/products/de/de/rmc/.

[70] J. Koskela, "Software configuration management in agile methods," *VTT Publications*, no. 514, p. 54, 2003.

[71] K. Schwaber, *The Enterprise and Scrum*. Microsoft Press, 2007.

[72] P. Abrahamsson, J. Warsta, M. T. Siponen, and J. Ronkainen, "New directions on agile methods: a comparative analysis," in *Proc. ICSE '03*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 244–254.

[73] U. Kelter, M. Monecke, and M. Schild, "Do we need 'agile' software development tools?" in *Revised Papers from the NetObjectDays*. London, UK: Springer, 2003, pp. 412–430.

[74] G. G. Miller, "The characteristics of agile software processes," in *Proc. TOOLS39*. Washington, DC, USA: IEEE CS, 2001.

[75] M. Laanti, O. Salo, and P. Abrahamsson, "Agile methods rapidly replacing traditional methods at nokia: A survey of opinions on agile transformation," *Inf. Softw. Technol.*, vol. 53, pp. 276–290, 2011.

[76] L. Jiang and A. Eberlein, "Towards a framework for understanding the relationships between classical software engineering and agile methodologies," in *Proc. APOS '08*. New York, NY, USA: ACM, 2008, pp. 9–14.

[77] J. Eckstein, "Scaling agile processes: Agile software development in large projects," in *Proc. XP/Agile Universe '02*. London, UK: Springer, 2002, pp. 279–280.

[78] J. E. Hannay, T. Dybå, E. Arisholm, and D. I. K. Sjøberg, "The effectiveness of pair programming: A meta-analysis," *Inf. Softw. Technol.*, vol. 51, pp. 1110–1122, 2009.

[79] T. Dybå and T. Dingsøyr, "Empirical studies of agile software development: A systematic review," *Inf. Softw. Technol.*, vol. 50, pp. 833–859, 2008.