To Agile, or not to Agile:
A Comparison of Software Development Methodologies

Ruslan Shaydulin, Justin Sybrandt

April 26, 2017

Abstract

Since the Agile Manifesto, many organizations have explored agile development methods to replace traditional waterfall development. Interestingly, waterfall remains the most widely used practice, suggesting that there is something missing from the many "flavors" of agile methodologies. We explore seven of the most common practices to explore this, and evaluate each against a series of criteria centered around product quality and adherence to agile practices. We find that no methodology entirely replaces waterfall and summarize the strengths and weaknesses of each. From this, we conclude that agile methods are, as a whole, unable to cope with the realities of technical debt and large scale systems. Ultimately, no one methodology fits all projects.

1 Introduction

Agile software development, a term introduced in 2001 in famous Agile Manifesto [1], took the world by a storm and quickly became, if not the most popular then certainly the most fashionable, development style. However, there is a lot of misunderstandings surrounding the agile development style. As teams adopt and change this methodology they raise questions about the limitations [2] and applicability [3] of this approach.

In this paper we explore the state-of-the-art methodologies which are widely used by software companies today with a focus on how agile methods have been altered to facilitate the need for planning. We compare different software development practices, both agile and traditional, in an attempt to find an answer to the questions like "Why does the waterfall persist?" and "What makes popular agile methods popular?"

This report is structured in the following way: Section 2 consists of the historical idealogical foundation of agile development, describing of the most famous agile method, extreme programming, and a quick discussion of architectural agility. Section 3 describes the evaluation methodology that we used to compare various methods and the rationale behind our evaluation framework. Section 4 contains descriptions of each studied methodology as well as short discussions on how they function in real-world circumstances. This section includes the evaluation results of seven of the most popular software development methods. Finally, ins section 5 we draw conclusions and outline possible future work.

2 Agile

In this section we will describe the original agile approach including its origins and its current state. This will act as a baseline on which we will compare modern reformulations of these core principles. Furthering this baseline we discuss Extreme Programming and Architectural Agility. The former is the principle agile methodology, the latter is a more modern approach with attempts to incorporate architectural design into the process. Although neither method is extraordinarily popular today, these serve to bookend the extremes of the agile development described in latter sections.

2.1 Agile: Basics

Martin, one of the authors of the original Agile Manifesto, begins his book on Agile Software development by defining agile as "the ability to develop software quickly, in the face of rapidly

changing requirements" [4]. This approach is motivated by what he calls the "runaway inflation" of the software development processes. This inflation begins when a team fails to deliver a product on time, and begin over-engineering a methodology to attempt to improve their performance. If the new methodology also leads to a failed project, an even more complicated replacement may be put in place. From there the cycle continues. As a result of this inflation, many companies were stuck in a quagmire of increasingly complex processes. A group of industry experts met and decided to create a set of values and principles that would help companies overcome these problems calling themselves the "Agile Alliance" [5].

There are many methods developed around the principles described in the agile manifesto, and it is important to separate the principles of agile from the methods and practices of agile software development (e.g., SCRUM). Failing to make this separation often leads to blindly following the word of practices of a certain method and ignoring the spirit of them. This can result in dangerous and inefficient cargo-cult mentality and in failure to deliver a good product. As a baseline methodology, we begin by exploring extreme programming (XP) before moving onto the seven most popular modern design methods.

2.2 Extreme Programming

Extreme programming is a method that is built around customers interacting closely with the development team throughout the course of the project [6, 4]. Customers see the system come together. As they learn more about what they need, customers and developers work together to adjust system requirements. To incorporate this flexibility, instead of producing detailed requirements, customers create user stories. These are simply a few words on an index card that summarize the rough idea of a requirement. As the system comes together, the customer sees the progress of the project and adjusts his or her expectations accordingly.

Development in XP moves in short cycles, and at the end of each iteration a system is shown to the customers in order to get their feedback. The average cycle lasts only two weeks long and consists of writing just enough code, typically in pairs, until a set of tests pass [7]. Speed is paramount, and code is integrated several times a day. However, the project is treated as a marathon and not a sprint, and therefore developers are not allowed to work overtime [6].

XP purposely ignored the practice of architectural design, and instead insists upon a minimally viable architecture. It is considered essential to avoid unnecessarily complex designs and resist adding infrastructure before it is strictly needed [8].

Workload is managed in XP through a series of user stories and story points [9]. At the beginning of each iteration a set of stories is created. Then they are assigned points that measure the cost of their implementation. The stories are broken into tasks that should take 4-16 hours for a developer to implement, after which each developer gets to pick tasks according to his or her budget. These budgets are determined by how many story points each developer completed during the last iteration. In a similar fashion, the total cost of user stories is determined by the number of story-points completed on the previous iteration. In this manner, XP is able to iteratively refine the cost and time estimate for a complex software project.

2.3 Architectural Agility

Enabling Agility Through Architecture

Brown et al. describe a method to produce "just-in-time" architecture designs while following an agile development process [10]. In their new method, "Architectural Agility," they modify the typical agile iteration process two introduce some forward-thinking elements. In an effort to improve the "Enhancement Agility," the speed at which a software team can introduce new features, Brown introduces project introspection to the agile iteration feedback loop. In a sense, this expands upon the iterative budget refinement present in XP, but focuses on technical debt. While practicing architectural agility, a team should conduct real options analysis and technical debt management at the start of each new iteration. This process begins with a technical research phase wherein potential future features are proposed and explored. This task is only intended to quickly identify routes the project might soon take, and produces a list of potential features and the architectural components they depend on. Then, real options analysis evaluates both the cost of taking actions in the upcoming iteration to facilitate those new features, as well as the potential expense of putting off the decision. Afterwards, technical debt management involves determining the cost of making modifications to the current codebase without major refactoring.

The end goal of architectural agility is an increased focus on the dependencies between user stories. This begins with an additional category added to the traditional agile release planning board, and ends with a dependency structure matrix. By focusing on these dependencies, as well as quality attributes, a team should be able to quickly identify the software components which are most central to the proposed system. This allows a team to understand which elements require fleshed-out architectures in the upcoming iteration.

Ultimately, it would seem that the project management structure proposed by Brown et. al. attempts to find a middle ground between the speed of agile, and the patience of waterfall. By taking the time each iteration to look a couple steps ahead, this method should be able to improve agile teams with complex software architectures.

3 Evaluation Methodology

In the following sections we detail the basis for our methodology evaluation and go on to explain our approach and the specifics of our evaluation criteria.

3.1 Literature review

In [11] Kitchenham describes the DESMET evaluation methodology. This method focuses on software development from multiple angles, taking input from surveys, case studies, and actual experiments. Both quantitative and qualitative results can be extracted from these three data sources. For example, case studies can both show the exact process as well as the opinions of experienced developers. Unfortunately, the DESMET method only works when a department is able to control their development process to ensure valid results, and therefore, makes the methodology difficult to apply in many agile environments. Still, we can borrow a similar data gathering approach our own evaluation methodology.

In [12], Sorensen compares waterfall software development models with incremental ones such XP and other agile methods. Though he shows many similarities, such as the fact that both allow for work force specialization, Sorensen highlights the ability of incremental methodologies to start out without clear set of requirements, as opposed to waterfall which requires the full set of requirements at the outset. Additionally, he notes that incremental methods require clean interfaces between modules while the waterfall approach is capable of maintaining more complex module connections.

In [13] Thayer, Pyster, and Wood detail twenty problems that many software projects face during development. Although this work was published in 1981, it is clear that most issues found on this list still plague a plethora of modern projects. For instance, these questions focus on topics such as unclear project requirements, poor project quality, and poor project estimates. We also note that while the first ten questions focus on the software itself, it remaining ten focus on a variety of important tangential topics, including management and testing. This, to us, highlights the importance of addressing non-functional qualities of the software creation process.

In [14], Awad compares what he calls "light" agile methods with "heavy" waterfall-like methods. For example, he points out that heavy approaches favor a "Command-Control" culture, whereas agile approaches tilt toward a culture of "Leadership-Collaboration". The same question of requirements, discussed in [12], is described in terms of "minimal upfront planning" and "comprehensive planning." In his work, Awad attempts to describe the balance between both approaches and shows that agile works well for projects that allow for inexpensive refactoring.

3.2 Our approach

Following approach introduced by Awad, we attempt to find a balance for the methodologies we encounter. In order to do so, we extend the set of project characteristic discriminators, introduced by [14]. Additionally, we adapt and utilize the questions asked by Thayer et. al. in order to evaluate how different software methodologies and architectural frameworks address a common set of concerns [13]. We note that the exact wording of these questions has, in some instances, gone out of date. Others simply lie too far outside the scope of a software development methodology to be applicable. For these reasons, we restrict the set of initial questions to a subset of topics. These topics are explained in detail below. For each evaluated methodology, we describe if and how it addresses these nine topics. We hypothesize that many will not strictly bind a development team

to a specific organizational structure or design selection, but instead will instead influence these topics in an indirect manner. For example, traditional agile with scrum [15] tends to produce very "wide" team hierarchies where all developers have very similar standings when making decisions. Additionally, we evaluate the agility of each methodology in an effort to identify if a strong correlation exists between software quality and agility. We do so in a similar manner, also described in the following section.

3.3 Methodology Description

Using sources and methods described above, we created the following evaluation methodology matrix. We define our own set of criteria which any good software development strategy should fulfill.

3.3.1 Process Quality Evaluation Criteria

1. **Requirements flexibility**: the method is able to respond to new and changing requirements.

2. **Requirements fulfillment guarantee**: method contains a mechanism to guarantee functional and non-functional requirements in order of their importance.

3. **Cost estimation**: the method has the ability to estimate the resources required to complete specific development tasks.

4. **Cost estimates refinement**: the method is able to update these cost estimates over time.

5. **Validation**: the method is capable of identifying when tasks are done properly vs when tasks are incomplete or contain errors.

6. **Quick validation**: the method is capable of quickly detecting when non-functional requirements are violated.

7. **Focus on customer**: the method focuses on client needs and ensures they are met continuously over the lifetime of the project.

8. **Understandability guarantee**: the method encourages the development of an understandable and maintainable code base.

9. **Technical debt control**: the method prevents runaway inflation of the cost of new features and bug fixes while controlling the amount of technical debt.

In addition to our quality criteria, we also present a series of criteria for evaluating the agility of a given software method. We do so in order to detect if there is a measurable correlation between method quality and method agility. This series of questions is directly inspired from the set of project characteristic discriminators defined in [14].

3.3.2 Process Agility Evaluation Criteria

1. **Prioritizes added value** The method prioritizes added value over nonfunctional benefits, refactoring, or documentation.

2. **Allows partial requirements** The method does not require stable or fully defined requirements.

3. **Focuses of small teams** The method emphasizes small "wide" teams over large hierarchical ones.

4. **Develops minimal viable architecture** The method does not develop software architecture beyond what is needed for the current working set of software requirements.

5. **Produces minimal documentation** The method does not produce much documentation, and the documentation which is produced is not intended to last the lifespan of the project.

6. **Relies heavily on customer feedback** The method includes customers at multiple points in the design process, as opposed to a single requirement gathering phase.

7. **Susceptible to unforeseen risks** The method is weak to unforeseen major risks or complications which were not foreseen. Waterfall, for example, spends substantial time documenting these risks and is therefore less susceptible.

4 Evaluated Methods

Vijayasarathy et. al. in [16] conducted a survey in which they asked 153 developers to describe their software development process. One of the most important results of this study is a listing of the most commonly used software development practices as of 2016. We use these results to inform our methodology reviews, allowing us to review the most widely used methodologies. The list of software methodologies which address in later sections follows the same order of popularity.

1. Waterfall

2. Agile Unified Process

3. Scrum

4. Test-Driven Development

5. Rapid Application Development

6. Joint Application Development

7. Feature-Driven Development

4.1 Waterfall

In [17], Royce describes the process which would eventually become known as waterfall development. This method includes seven steps which are completed in order and should result in stable software as a result. These steps include, system requirements, software requirements, analysis, program design, coding, testing, and operations. Royce notes that these steps do not typically form a linear flow from one to the next, instead many backward pointing edges between these states may exist. For example, as testing reveals bugs, a project might need to return to the program design phase. Additionally, these main phases represent sub-trees within them selves. For example, Royce points out that whole departments might be assigned to just the testing phase.

Waterfall typically requires a long time of requirements gathering and project planning before any code is written. This process attempts to identify the full set of project requirements. Other methods, such as [18] by Benington and [19] by Ramamoorthy also incur a large up front requirement gathering cost and were published around time same time. Since then, numerous other flavors of waterfall development have been introduced such as the iterative model described in [20], which closely resembles the "iterative waterfall method."

It is unlikely that those survey respondents who reported using waterfall in 2016 followed the original specification without modification [16]. Waterfall has changed a lot over the years as developers have began experimenting with a more iterative approach. For the purposes of our analysis, we consider all of the above waterfall variants together, including iterative agile. As a whole, these methods are characterized by large upfront costs in requirement gathering, inexpensive bug fixing, accurate time estimates, and inflexibility to changing or uncertain requirements. It seems clear that there are still many projects which are sufficiently well-defined to facilitate such a starched methodology.

4.2 Agile Unified Process

Agile Unified Process was developed by Scott Ambler [21] as a simplified version of the Rational Unified Process. AUP development was stopped in 2006. In 2009 Disciplined Agile Delivery was introduced to supersede it, yet the survey results in [16] make it clear that AUP remains very popular.

AUP applies agile techniques, such as Test Driven Development (TDD) and Agile Model Driven Development (AMDD) to improve the productivity of a team. It acts as a unifying framework for multiple agile methodologies and processes.

The AUD approach is "serial in the large" and "iterative in the small". It consists of four subprocesses or work flows: Modeling, Implementation, Testing and Deployment, each of which goes through four phases: Inception, Elaboration, Construction and Transition.

The principles of AUD emphasize simplicity. The simplest possible tools are preferred to complex products, minimal viable documentation is preferred to detailed documentation. It's aiming to prioritize high-value activities over trying to define everything that can possibly happen in a lifespan of a project.

Success of Agile Unified Process over other agile methodologies can be attributed to its flexibility, as well as to its simplicity.

4.3 Scrum

Scrum [15] is one of the most popular frameworks for implementing agile. Its defining characteristic is commitment to short iterations of work.

In scrum, a product is developed through a series of fixed-length iterations, called sprints, that allow for software updates at a regular cadence. The part of scrum that is the most attractive to a team is the idea that some call "continuous inspiration" [15]. Team members are motivated by tangible, visible progress at the end of each sprint, as well as by the ability to "show off" during the sprint demo.

Sprint consists of four "ceremonies" [22]:

1. Sprint planning – team meeting where the next sprint is outlined.

2. Daily stand-up – daily 15 minute meeting for the team to sync up.

3. Sprint demo – weekly meeting where teammates showcase what they will ship during the week.

4. Sprint retrospective – weekly analysis of what went wrong and what went right during the previous week.

Scrum has three specific roles: product owner, scrum master, and the development team. The product owner works with the business requirements and gives requirements to the team. The scrum master coaches the team and make sure the team observes scrum practices. The development team works closely together in an open and collaborative manner, talking regularly at scrum meetings. A potential point of failure, of course, comes from the quality of the product manager who is the sole point of contact between the development team and the customers.

4.4 Test Driven Development

Test Driven Development (TDD) is a design process which focuses on developing tests for "units' before the implementation of the functionality in those units [23]. This practice creates a sort of programmers metacognition, where a significant portion of the development process centers on analyzing code deliverables. In relation to other methodologies discussed in this section, TDD relies the most heavily on specific testing technologies. For example, TDD would not be where it is today if it were not for automatic testing frameworks such as JUnit and ANT. Additionally, TDD often plays a secondary role in other agile methodologies such as Extreme Programming [6]. This is primarily because TDD alone fails to cover the entire software design process, and instead focuses almost entirely on the technical aspects of writing code. After-all, how would one write a test to guarantee a quality attribute such as "discoverability" or "flexibility." Nevertheless, TDD has recently grown into its own software design method [24].

In [25], IBM employees address the efficacy of TDD in a large scale software system, JavaPOS. The IBM group noticed a higher defect rate than desired, so switched to a TDD-centric approach in an effort to improve their product. They did some initial design work with UML diagrams and a prototype, which they were able to test against the existing JavaPOS standard. As a result of this experiment, IBM noticed a 50% reduction in error rate, a slight decline in productivity, and a more flexible design process.

A more recent study [26] looks to explore the effect TDD has in the modern software production process. By aggregating over a thousand papers evaluating TDD, Bissi et. al. note that most TDD studies report an increase in code quality, and a decrease in productivity. Interestingly the study also found that almost every TDD project was written in C++ or Java. This speaks to the limited scope of TDD, and how tied it is to specific testing frameworks.

4.5   Rapid Application Development

Rapid Application Development (RAD) emphasizes user involvement in every step of the design process and was initially proposed by Martin in [27]. In this method, software design occurs in four distinct phases: the requirements planning phase, the user design phase, the construction phase, and the cutover phase. These phases often follow the typical systems development life cycle [28]. In the requirements fathering phase all stakeholders, especially key users, are brought together to determine system requirements, like many other methodologies discussed here. What RAD does differently, is that user interaction continues into the design phase. As opposed to traditional agile methods, projects following the RAD method will prioritize prototyping over partially-functional deliverables. This means that prototypes can emerge as early as architecture planning time. By doing so, RAD ensures that user feedback can be incorporated as the system early and continuously. Additional benefits such as risk control, and budget sensitivity have also been noted as side-effects to the heightened sense of user involvement [6]. Although it has also been noted that the rapid pace of prototyping can result in poor design as developers seek short term functionality while ignoring technical debt [29, 30].

A notable and interesting result of this methodology is the quantification of the user. By giving users so much power in the software design process, it can become ambiguous how to proceed when users inevitably disagree. Those practicing this methodology typically separates users into various roles, one of the most common being that of *visionary* (one who instigates the project), *ambassador* (one who represents the user community at large), and *advisor* (one who is asked to influence design decisions) [31]. Although such an approach can clarify many typical crossing points, arguably this highlights another weakness in the RAD development process. RAD results seem to be limited by the quality and commitment of the users a software team has access to.

4.6   Joint Application Design

Joint Application Design (JAD) is a methodology for operationalizing user involvement and user participation[32]. Is assumes that the final quality of the software is proportional to the degree of final user involvement in the software's development[33]. While some argue that this notion is unsupported and lacks any evidence [34], there is a clear intuition that the best way to understand the user's needs is to involve him or her in the development process and adjust the requirements as user's objectives change. JAD takes this approach to the extreme.

JAD is centered around a structured meeting, called the "session," which the entire software development process resolves around. During the session, users and software developers discuss a clearly formulated agenda in an organized fashion. In some approaches the meetings are less structured and prefer the "free discussion" approach, however they have proved to suffer from a number of problems[35]. This effort maintains contact between the development team and software end users throughout the lifespan of the software process.

JAD meetings have proved to be superior to interviewing techniques employed by other methods [36] and is widely used in industry and even attracted the interest of the scientific community.

It is important to point out that JAD is nothing more than a technique for getting the requirements from users. It can, for example, be integrated into the Agile Unified Process. It, by no means, is a full substitute for a method like waterfall; it has to be complemented with a way to formalize the process of the actual application development.

4.7   Featured Driven Development

Feature Driven Development (FDD) follows along with many other agile methods in that it focuses on short value-adding sprints and attempts to introduce new working functionality in each iteration [37]. What sets feature driven development apart is the speed and granularity of each sprint [38]. Each feature is described as a short sentence containing a clear action, result, and object. For example a feature may be as short as "display the shopping cart to the customer." These features

are grouped into sets, and each set describes a two week iteration [39]. FDD promotes very modular architectures, as each feature should, in a perfect world, be concurrently developed.

Typically, FDD start with UML modeling. As features are planned, the necessary software components and their connections should grow organically. Arguably, this is the opposite of typical architectural design processes where a large system is split into encapsulated submodules. Additionally, FDD defines a hierarchy of developers, appointing some to be "chief programmers" responsible for small teams and planning sprints [14]. These leaders act as a way to fluidly create and dismantle teams around features, as well as to regulate the quality of software produced in an iteration. Interestingly though, as opposed to many other agile methods, FDD actively discourages refactoring. Instead the development cycle focuses on added value to an extreme. After-all once a feature is present in the software, the customer is satisfied [40].

It would seem that FDD is a very natural, albeit naive, approach to software design. If a team is good and spends adequate time each iteration on the closing code review, it is possible that code quality may not deteriorate too quickly. It does appear that the method requires substantial upfront cost in the FDD version of requirements gathering in order to put together an initial plan that will not change too drastically during the lifetime of the project, otherwise an FDD project would require deep refactoring. Lastly, FDD fails to account for many non-functional qualities which typically depend on interactions between features. For example, FDD could easily produce a system had to both produce a shopping cart as well as track purchasing data, but would be ill equipped to ensure a latency requirement between user actions and the tracking system.

		Methodology						
		Waterfall	AUP	Scrum	TDD	RAD	JAD	FDD
Quality Criteria	Requirements flexibility	No	Yes	Yes	Yes	Yes	Yes	No
	Requirements fulfillment guarantee	Yes	Yes	Yes	No [1]	Yes	No	Yes
	Cost estimation	Yes	Yes	Yes	No	Yes	No	Yes
	Cost estimates refinement	No	Yes	Yes	No	Yes	No	Yes
	Validation	Yes	Yes [2]	Yes [3]	Yes	Yes	Yes	Yes
	Quick validation	No	Yes [2]	Yes [3]	Yes	Yes	Yes	Yes
	Focus on customer	No	Yes [4]	Yes	No	Yes	Yes	No
	Understandability guarantee	Yes [5]	No	No	No	No	Yes [6]	No
	Technical debt control	Yes	No	No	Yes	No	No	No
Agility Criteria	Prioritizes added value	No	Yes	Yes	Yes	Yes	Yes	Yes
	Allows partial requirements	No	Yes	Yes	Yes	Yes	Yes	Yes
	Focuses on small teams	No	Yes [7]	Yes	Yes	Yes	Yes	Yes
	Develops minimal viable architecture	No	Yes	Yes	Yes	Yes	Yes	Yes
	Produces minimal documentation	No	Yes	Yes	Yes	Yes	No	Yes
	Relies heavily on customer feedback	No	Yes	Yes	No	Yes	Yes	No
	Susceptible to unforeseen risks	No	Yes	Yes	Yes	No	Yes	Yes

Table 1: Results of methodologies evaluation

5 Conclusion

In this paper, we have presented a comparative analysis of a number of the most popular software development methodologies. The results show that each of the "agile" methodologies has an important basic criteria that it doesn't cover, such as technical debt control or cost estimation. In smaller projects these issues can be ignored since the costs are small and the potential for technical

---

[1] Only Functional
[2] via TDD
[3] via weekly demos
[4] via AMDD
[5] Accomplished with lengthy documentation and planning.
[6] Via continuous user engagement in the development process
[7] Works for all sizes

debt is limited. For really large and complex projects, these limitations render the "agile" methods inapplicable.

Therefore, we believe that these innate qualities of "agile" methods make them applicable only for a certain subset of the problems: small and solvable in small teams. However, the agile methods that we've looked at do not scale well. For large and sprawling projects, a paradigm that over=-emphasizes many person-to-person interactions stops working.

We find that these limitations explain the persistence of the traditional waterfall approach. Despite being bulky and prone to generating redundant documentation, it is a tried and true method that has proved to be able to deliver working products. It is understandable that many managers decide to err on the side of caution, especially considering the fact that benefits of using many agile methods are at best unclear.

We find it unsurprising that Vijayasarathy noticed over 45% of software teams are using a hybrid approach [16]. As we show in our table above, no one methodology properly addresses all important aspects of the software design process. By using a combination of approaches, greater results can be achieved. For example by using waterfall for project planning, scrum for short term goals, and TDD for a guarantee on software correctness, a team can mitigate the downfalls present in each method while highlighting their strengths.

References

[1] Agile Manifesto. [Online]. Available: http://agilemanifesto.org

[2] D. Turk, R. France, and B. Rumpe, "Limitations of agile software processes," *arXiv preprint arXiv:1409.6600*, 2014.

[3] A. Qumer and B. Henderson-Sellers, "An evaluation of the degree of agility in six agile methods and its applicability for method engineering," *Information and software technology*, vol. 50, no. 4, pp. 280–295, 2008.

[4] R. C. Martin, *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[5] A. Alliance, "Agile manifesto," *Online at http://www. agilemanifesto. org*, vol. 6, no. 1, 2001.

[6] K. Beck, *Extreme programming explained: embrace change*. addison-wesley professional, 2000.

[7] ——, "Embracing change with extreme programming," *Computer*, vol. 32, no. 10, pp. 70–77, 1999.

[8] M. C. Paulk, "Extreme programming from a cmm perspective," *IEEE software*, vol. 18, no. 6, pp. 19–26, 2001.

[9] K. Beck and M. Fowler, *Planning extreme programming*. Addison-Wesley Professional, 2001.

[10] N. Brown, R. Nord, and I. Ozkaya, "Enabling agility through architecture," DTIC Document, Tech. Rep., 2010.

[11] B. A. Kitchenham, "Evaluating software engineering methods and tool part 1: The evaluation context and evaluation methods," *ACM SIGSOFT Software Engineering Notes*, vol. 21, no. 1, pp. 11–14, 1996.

[12] R. Sorensen, "A comparison of software development methodologies," *CrossTalk*, vol. 8, no. 1, pp. 10–13, 1995.

[13] R. H. Thayer, A. B. Pyster, and R. C. Wood, "Major issues in software engineering project management," *IEEE Transactions on Software Engineering*, no. 4, pp. 333–342, 1981.

[14] M. Awad, "A comparison between agile and traditional software development methodologies," *University of Western Australia*, 2005.

[15] K. Schwaber and M. Beedle, *Agile software development with Scrum*. Prentice Hall Upper Saddle River, 2002, vol. 1.

[16] L. R. Vijayasarathy and C. W. Butler, "Choice of software development methodologies: Do organizational, project, and team characteristics matter?" *IEEE Software*, vol. 33, no. 5, pp. 86–94, 2016.

[17] W. W. Royce *et al.*, "Managing the development of large software systems," in *proceedings of IEEE WESCON*, vol. 26, no. 8.   Los Angeles, 1970, pp. 1–9.

[18] H. D. Benington, "Production of large computer programs," *Annals of the History of Computing*, vol. 5, no. 4, pp. 350–361, 1983.

[19] C. Ramamoorthy, W. Tsai, T. Yamaura, and A. Bhide, *METRICS GUIDED METHODOLOGY*.   IEEE, 1985, pp. 111–120.

[20] N. M. A. Munassar and A. Govardhan, "A comparison between five models of software engineering," *IJCSI*, vol. 5, pp. 95–101, 2010.

[21] S. Ambler, *Agile modeling: effective practices for extreme programming and the unified process.* John Wiley & Sons, 2002.

[22] S. Alliance, "Learn about scrum," *Scrum Alliance*, 2016.

[23] D. Janzen and H. Saiedian, "Test-driven development concepts, taxonomy, and future direction," *Computer*, vol. 38, no. 9, pp. 43–50, 2005.

[24] L.-O. Damm, L. Lundberg, and D. Olsson, "Introducing test automation and test-driven development: An experience report," *Electronic Notes in Theoretical Computer Science*, vol. 116, pp. 3–15, 2005.

[25] E. M. Maximilien and L. Williams, "Assessing test-driven development at ibm," in *Software Engineering, 2003. Proceedings. 25th International Conference on.*   IEEE, 2003, pp. 564–569.

[26] W. Bissi, A. G. S. S. Neto, and M. C. F. P. Emer, "The effects of test driven development on internal quality, external quality and productivity: A systematic review," *Information and Software Technology*, vol. 74, pp. 45–54, 2016.

[27] J. Martin, *Rapid application development.*   Macmillan Publishing Co., Inc., 1991.

[28] I. Jacobson, G. Booch, J. Rumbaugh, J. Rumbaugh, and G. Booch, *The unified software development process.*   Addison-wesley Reading, 1999, vol. 1.

[29] A. Gerber, A. Van Der Merwe, and R. Alberts, "Practical implications of rapid development methodologies."   Computer Science and Information Technology Education Conference, 2007.

[30] P. Beynon-Davies, C. Carne, H. Mackay, and D. Tudhope, "Rapid application development (rad): an empirical review," *European Journal of Information Systems*, vol. 8, no. 3, pp. 211–223, 1999.

[31] H. Mackay, C. Carne, P. Beynon-Davies, and D. Tudhope, "Reconfiguring the user: using rapid application development," *Social studies of science*, vol. 30, no. 5, pp. 737–757, 2000.

[32] E. Carmel, R. D. Whitaker, and J. F. George, "Pd and joint application design: a transatlantic comparison," *Communications of the ACM*, vol. 36, no. 6, pp. 40–48, 1993.

[33] R. Hirschheim and M. Newman, "Symbolism and information systems development: myth, metaphor and magic," *Information Systems Research*, vol. 2, no. 1, pp. 29–62, 1991.

[34] B. Ives and M. H. Olson, "User involvement and mis success: A review of research," *Management science*, vol. 30, no. 5, pp. 586–603, 1984.

[35] E. J. Davidson, *An exploratory study of joint application design (JAD) in information systems delivery.*   Center for Information Systems Research, Sloan School of Management, Massachusetts Institute of Technology, 1993.

[36] S. A. Becker, E. Carmel, and A. R. Hevner, "Integrating joint application development (jad) into cleanroom development with icase," in *System Sciences, 1993, Proceeding of the Twenty-Sixth Hawaii International Conference on*, vol. 3.   IEEE, 1993, pp. 13–21.

[37] J. Hunt, "Feature-driven development," *Agile Software Construction*, pp. 161–182, 2006.

[38] M. Benoit, R. Anthony, and L. B. Wee, "Feature-driven development," 1999.

[39] S. R. Palmer and M. Felsing, *A practical guide to feature-driven development.* Pearson Education, 2001.

[40] S. Khramtchenko, "Comparing extreme programming and feature driven development in academic and regulated environments," *Feature Driven Development*, 2004.