5-2022

A Study of Software Development Methodologies

Kendra Risener
University of Arkansas, Fayetteville

A Study of Software Development Methodologies

An Undergraduate Honors College Thesis
in the
Department of Computer Science and Computer Engineering
College of Engineering
University of Arkansas
Fayetteville, AR

by

Kendra E. Risener
kerisene@uark.edu

April, 2022
University of Arkansas

Abstract

Software development methodologies are often overlooked by software engineers as aspects of development that are handled by project managers alone. However, if every member of the team better understood the development methodology being used, it increases the likelihood that the method is properly implemented and ultimately used to complete the project more efficiently. Thus, this paper seeks to explore six common methodologies: the Waterfall Model, the Spiral Model, Agile, Scrum, Kanban, and Extreme Programming. These are discussed in two main sections in the paper. In the first section, the frameworks are isolated and viewed by themselves. The histories, unique features, and professional opinions regarding the methodologies are explored. In the second section, the methodologies are compared to one another, particularly within the context of ideal development environments and methodology advantages and disadvantages. It becomes apparent that the Waterfall and Spiral models are immensely different from the Agile, Scrum, Kanban, and Extreme Programming methodologies. This is because the Waterfall Model and the Spiral Model are both software development life cycle models, which indicates that there is a specific flow and set of rules to follow when it comes to development. The other methodologies, however, embrace the idea of flexibility. The process of creating software often changes. For example, a client could deliver new requirements for the software after development has already begun. Thus, the Agile-based methods do not attempt to create a rigid step-by-step development plan. Rather, they incorporate the idea of embracing change, empowering teams to accept that their development plan will have to change often. This paper further explores these considerations with the intention of promoting an interest in learning about the best ways to go about planning and developing software.

Contents

1 Introduction

1.1 Background

Many software engineers tend to think about only the lower-level implementation of a project, which entails the actual process of developing software and, more generally, coding, but in a world with increasingly complex projects, engineers need to understand how to create software in an efficient manner. This requires thinking at a higher level about how designing and developing a system in a particular manner can be either conducive or harmful to efficient engineering practices. One of the ways software developers can develop software at a consistent pace that reduces time and money spent is by following popular, well-defined software development methodologies. Software development methodologies are a combination of both practices and values. The practices will help guide the developers on what they need to accomplish and when they need to accomplish it. The values serve as a simple ethical code that software engineers should follow. Due to this, understanding the main software development methodologies is an important endeavor. There are many articles and research papers done on various software methodologies. This paper will be a study of the general sentiments and ideas expressed in these resources in order to promote the research and consideration of development methodologies as a regular practice.

1.2 Content

This paper will discuss the general features of six popular development methodologies. This will include information that would guide an individual to both properly understand and follow a methodology, such as the specific practices, team roles, meetings, and values that accompany each method. In addition, to give more background into the methodologies, the general history will also be explored. Each methodology also has its own unique features and practices. These special features will be studied as well. While much of what will be mentioned in this paper is idealistic and theoretical, an effort to consider how methodologies actually work in the real world of software development will be made through discussing the general opinions industry professionals have regarding a particular methodology. Upon considering these varying opinions, one may find that a development methodology may not be as efficient as it theoretically should be, as different methodologies may thrive used in conjunction with different projects and environments.

In addition to analyzing the methodologies on an individual basis in the beginning of the paper, comparisons between the methods will be made. For example, the main development environments that suit each framework will be compared and contrasted. The advantages and disadvantages of the methodologies will also be discussed.

The development methodologies that will be discussed in-depth are the Waterfall Model, the Spiral Model, Agile, Scrum, Kanban, and Extreme Programming methods. Although the previously mentioned methods are often debated as to whether some of them can fully be considered methodologies, within the scope of this paper they will be referred to as both frameworks and methodologies. Frameworks are generally seen as light-weight methodologies, but there is much disagreement in the project management world about how and when to differentiate between frameworks and methodologies, and if this differentiation even matters or if it is merely semantics. Thus, for these reasons, the Waterfall Model, the Spiral Model, Agile, Scrum, Kanban, and Extreme Programming will all be considered both frameworks and methodologies in order to make reading this paper more straightforward.

Throughout the content of the thesis, there may be italicized text. It will appear as follows: *example italicized text.* Italicized text indicates that a word or group of words is a key term. Key terms will be explained in the glossary section towards the end of the paper. This will be done to prevent confusion.

2 Development Methodologies

This study will focus on exploring six software development methodologies: the Waterfall Model, Spiral Model, Agile, Scrum, Kanban, and Extreme Programming. These methodologies are considered to be some of the most well-known in the field of software engineering. In order to fully understand what following each of these frameworks entails, various aspects will be considered. Specifically, a general explanation of each methodology, the history behind it, its unique features, and general developer opinions of the methodology will all be discussed.

2.1 Waterfall Model

WHAT IS THE WATERFALL MODEL?

The Waterfall Model is a *software development life cycle (SDLC)* model. An SDLC model is a general software development framework or methodology which follows a specific set of steps in a specific order. For example, in this methodology, the development process is divided into different phases, and a developer can only proceed to the next phase when the current phase is complete. The phases do not overlap, making development flow sequentially. Due to this, the Waterfall Model is referred to as a *linear-sequential life cycle model.* Progress moves downward in the Waterfall Model, evoking the feeling of a waterfall. The water in a waterfall does not go back up; it plummets down toward the earth. The Waterfall Model is similar in that once the cycle starts, the developer has to continue until reaching the end; one cannot go back or flow upwards to return to previous phases. The only way one can go back to a previous phase is by starting the development cycle over again.

There are slight variations of the Waterfall Model's phases. However, the general consensus is that there are six phases. The first phase is Requirement Analysis. During this phase, the requirements of the project are discussed with the client. All requirements are analyzed in order to determine if there are any vague or inconsistent requirements. These issues are then discussed with the client, who agrees upon a solution with the development team. Once the requirements are finalized, the requirements for the *system* are noted in a *software requirement specification (SRS) document.* This formal *documentation* serves as an agreement between the team and the client, outlining the official specifications of the project.

The second phase is System Design. The development team utilizes the SRS document and analyzes it to create a design that fulfills all of the client's requirements. This phase involves designing the software and specifying the hardware used to develop the system.

The third phase is Implementation. The software that follows the design formulated in the previous phase is created at this time.

Once the Implementation phase is complete, the fourth phase begins, which is Testing. There are various tests that can be performed on the newly-created software. Some of the common tests for software include *unit testing*, *integration testing*, and *acceptance testing*. Unit tests are done for each block of code with unique functionality in the system. Integration tests combines the individual functionalities and tests the system as a whole. Acceptance testing is done by the client. During this process, the client performs the final tests and decides to accept or reject the software. If the software is rejected, the team will have to restart the Waterfall Model at the first phase and make changes to the software requirements.

The fifth phase is Deployment. The software product is released into the real world according to the client's release requirements. For example, the software could be released for the client's company to use only, or it could be released on the internet, under the name of the client's company.

The sixth and final phase is Maintenance. After deploying the software, there are potential changes that will need to be made. Once the updates are complete, the new software is then released once more. There are different types of maintenance that can be done on the software. The three main types of maintenance are *corrective maintenance*, *perfective maintenance*, and *adaptive maintenance*. Corrective maintenance is done to correct issues that were not discovered during testing. Perfective maintenance is maintenance that enhances the existing functionalities of the product. Adaptive maintenance occurs if the software needs to be moved to a new environment.

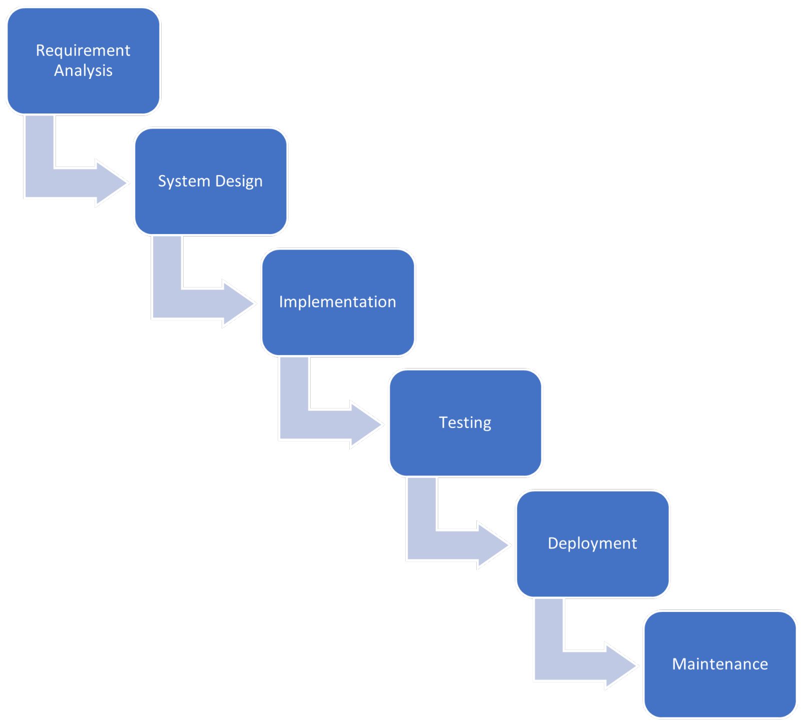

Figure 2.1.1: *The phases of the Waterfall Model*

HISTORY

The Waterfall Model, in modern times, was first formally defined by Winston Royce in 1970. Royce, a computer scientist working in the software engineering industry, was privy to the different development styles of complex systems. He wrote a paper called "Managing the Development of Large Software Systems." In this paper, he outlined the Waterfall Model, but he did not endorse following it. He said in this very paper that "I believe in this concept, but the implementation above is risky and invites failure" [35]. Despite his disapproval, the article gained traction for the Waterfall Model because of how simple Royce made the methodology sound.

SPECIAL FEATURES

The Waterfall Model is known for its simplicity. One does not need a certification or high level of mastery in order to implement this methodology. This is due to the fact one only has to follow the six phases in order to correctly apply the Waterfall Method. There are not any underlying principles or attitudes such as communication, respect, or adaptability that need to be implemented like these are in the other methodologies that will be discussed. Thus, the Waterfall Model is not really a philosophy or general mindset for creating software; it is simply a set of steps a team must follow. Because of this rigidity, the Waterfall Model does not garner the same excitement and attention as the *Agile-based methods*.

GENERAL PROFESSIONAL OPINIONS

The Waterfall Model serves as a basis for many other software development life cycle models. However, after consulting different resources, the consensus appears to be that the Waterfall Model is flawed. The Waterfall Model does not work well for long, complex projects. This is because there is little room for adjustments. When the development teams need to go back and fix an aspect of the software, the Waterfall Model has to be either completed until the end of the sixth phase or has to be restarted entirely. This is unrealistic; a client could change the software requirements in the middle of a project, or it could be discovered that the original software design is not very efficient. There are many such scenarios in which the software development process would require change. The Waterfall Model does not handle the reality of an ever-changing world of technology. Another unrealistic aspect of the Waterfall Model is the idea that the phases cannot overlap with one another. The overlapping of phases can potentially reduce the time of development and the overall cost of the project. Working on different phases at once simply should not be prohibited in the real world. Despite the previously mentioned disadvantages of following the Waterfall Model, there are advantages to using this development method as well. One of the biggest appeals of the Waterfall Model is its simplicity. Many development methodologies today have many different ceremonies, philosophies, and practices that have to be followed in order for a development team to be implementing the methodology as it was intended. However, with the Waterfall Model, one only has to follow the six basic development phases of Requirement Analysis, System Design, Implementation, Testing, Deployment, and Maintenance. In addition, the Waterfall Model focuses on having well-documented requirements, as evidenced during the Requirement Analysis phase when an official software requirement specification document is completed. When a software system has documented requirements, it is easier for the *developers* to see the overall goal of their project and understand what is needed of them in the design and implementation processes.

2.2 Spiral Model

WHAT IS THE SPIRAL MODEL?

Similar to the Waterfall Model, the Spiral Model is a software development life cycle (SDLC) model. The spiral shape is a way to visualize the way development should occur under this methodology; it should loop until the end of the project when all of the client's requirements are addressed. There are four main phases in the Spiral Model.

The first phase is Planning. This is where communication with the customer occurs to ensure that the requirements are understood by the development team. The development team will then take this requirement information and create a general development plan that addresses basic concerns such as the technologies that should be used, the delegation of tasks, and budgetary and time guidelines.

In the second phase, Risk Analysis and Mitigation, the developers work together to discuss the potential *risks* associated with this project. They also develop possible solutions to these risks. By the end of this phase, a *prototype* is created. The prototype serves as a basic model for the finalized software. By creating a prototype that can be shown to the customer, the development team is further mitigating risks and can gain either approval or disapproval from the client for the current design.

The third phase is Development. During this phase, actual software is produced. The software engineers will take the feedback they received from the prototype and create software that fulfills the customer's requirements.

Lastly, the fourth phase is Review and Evaluation. The team will test the software they created. After the software has been tested and generally approved by the team, the customer reviews the resulting product. The customer's feedback from this phase is used to determine the Planning and Risk Analysis and Mitigation phases of the next round. However, if the customer indicates during this fourth phase that the product is completed to their standards, then the development cycle is complete.

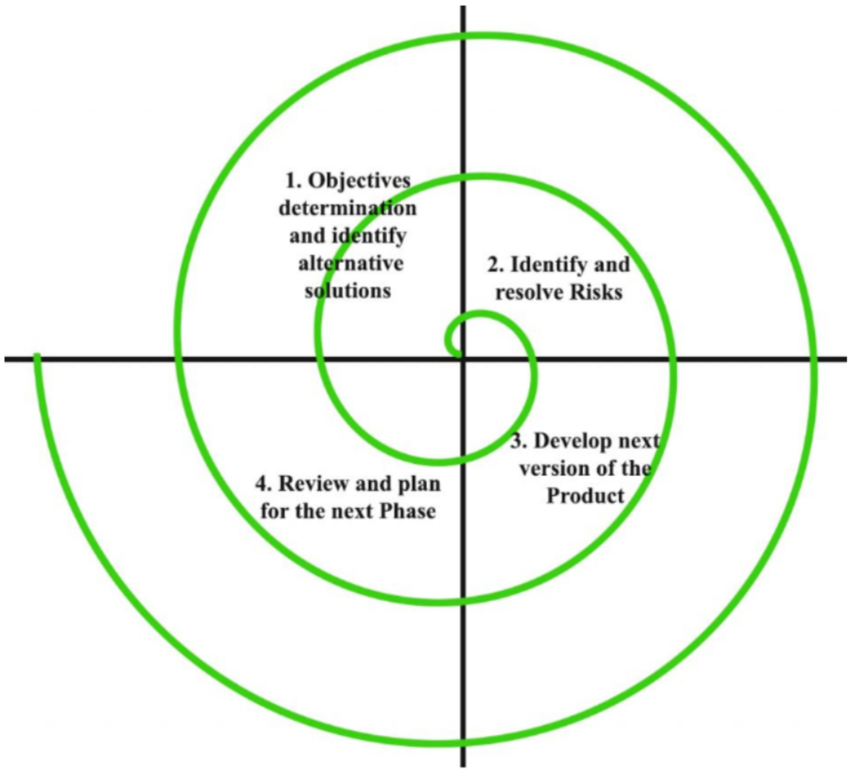
1. Objectives determination and identify alternative solutions
2. Identify and resolve Risks
3. Develop next version of the Product
4. Review and plan for the next Phase

Figure 2.2.1: *The development cycle of the Spiral Model* [36]

HISTORY

The creation of the Spiral Model is relatively easy to trace compared to the other methodologies that were developed more naturally in either the software engineering or manufacturing fields. It was developed by Barry W. Boehm, a well-accomplished software engineer, in 1988 in his published article entitled "A Spiral Model of Software Development and Enhancement". He created the Spiral Model as a solution to the drawbacks of the Waterfall Model, which does not always consider project risks or changing requirements until close to the completion of the project. He proposed the Spiral Model as an efficient risk-driven approach to software development. After Boehm developed the Spiral Model, he conducted research on its efficacy. He created software productivity system software, and he found that the metrics recorded by the software indicated that projects that used the Spiral Model increased efficiency. In his research paper, Boehm claims, "All of the projects fully using the system have increased their productivity at least 50 percent; indeed, most have doubled their productivity (when compared with cost-estimation model predictions of their productivity using traditional methods)" [7]. The methods Boehm is referring to in this quotation are the Waterfall Model and code-and-fix model, which is simply coding and fixing any mistakes that may arise later. Boehm's knowledge of software development methodologies comes from his extensive experience in the software engineering field. He went on to serve as the U.S. Department of Defense Director of the DARPA Information Science and Technology Office from 1989 to 1992. He also worked as Director of the DDRE Software and Computer Technology Office. He currently holds a distinguished professorship at the University of Southern California, where he also serves as the Director of the Center for Software Engineering.

SPECIAL FEATURES

The Spiral Model is unique because it emphasizes the importance of prototyping. This ensures that the client is able to give feedback on the general implementation of the project before the team has to entirely finish a product in which the client might be displeased. The prototypes made when a team follows the Spiral Model mean that there is opportunity for frequent feedback from the client. Communicating often with the client will greatly reduce the chance that a product in its final stages will need drastic changes. The Spiral Model also places high importance on handling risks. This means that the development team will have to really consider the possible risks and how to manage those risks before they even start developing software. This is different from other popular development methodologies because most processes only deal with risks as they occur–not before they occur. Overall, this can reduce the number of catastrophic mistakes that can arise during development. Another special feature of the Spiral Model is that it has only four phases. This makes it easier to both comprehend and implement correctly.

GENERAL PROFESSIONAL OPINIONS

The Spiral Model is similar to other software development models in the sense that there are clear advantages and disadvantages. According to most software developers, one key advantage is the presence of risk analysis. This can prevent major issues in the development phase. In addition, there is also rapid prototyping. This can serve as a great guide for the software team and the client alike; they are able to see what they would and would not like to see in the final product. Another advantage is that software is produced early in the life cycle of the Spiral Model. This means that there is faster feedback from clients, and software architecture can be changed more easily. Furthermore, another common advantage mentioned by developers is that there is a vast amount of project monitoring. Many developers like to jump in and start coding straight away. However, the Spiral Model has software engineers take a step back and really consider potential risks in different approaches and analyze different software architectures that could be utilized.

Despite the strong advantages of the Spiral Model, there are considerable weaknesses that many developers and project managers have found in this SDLC. For example, the Spiral Model can be an expensive methodology to follow. This is because prototypes and *deliverables* have to be made often for the client. This takes additional time, and it also takes time away from actually developing the product the client wants to see. Another disadvantage of the Spiral Model is that the development team has to be knowledgeable enough to recognize and fix risks beforehand. If the team is working with a new technology, they may not be aware of potential risks and rush into the project in spite of the protective measures the Spiral Model takes against risks.

2.3 Agile

WHAT IS AGILE?

Agile was developed to enable software engineering teams to regularly deliver high-quality *executables* within budget and on time. Agile is comprised of various development methods that have emerged through the years. It is a broad development methodology; it serves as the basis for many popular, more specific methodologies like Scrum, Extreme Programming, and Kanban, which will be discussed later in this paper. Agile is more of a mindset. It doesn't give hard and fast rules about how one should develop software. Despite this, there are a few general suggestions that the official Agile documentation outlines [14]. In an Agile environment, there should be small, spread-out investments instead of large investments in the beginning of development. In addition, project managers should direct their software engineers to focus on projects and tasks that promote the reputation and revenue of the company; this ultimately means that focusing on client satisfaction is highly important. Also, in an Agile environment, there should be more communication between software team members and business team members. This ensures that the client is truly getting what they want, and it also creates a more dynamic environment that limits the number of problems caused by a lack of communication. The Agile methodology also instructs those in charge to place trust in staff to work and complete high-quality software. Lastly, one of the most important tenets of Agile development is being able to adapt. Agile embraces the changing world of technology; thus, Agile methods promote more flexibility. Agile wants both software and people that are able to adapt to changes in technology, the company, and the world.

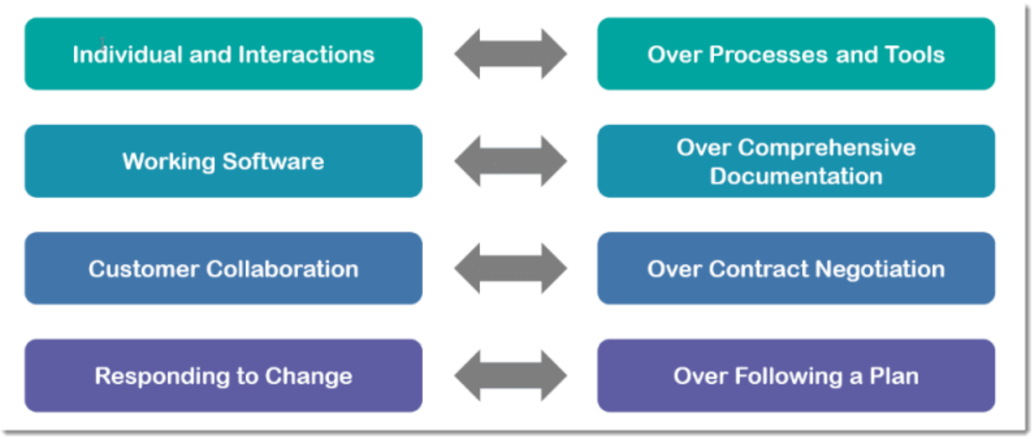

Figure 2.3.1: *Main Principles behind Agile* [37]

HISTORY

Agile-like principles certainly existed before 2001, but they had never really been written down or impacted the software world as much as when the Agile Manifesto was released in that year [14]. The Agile Manifesto gave individuals a specific set of principles and values to follow in order to improve their final products. This Agile documentation came about in 2001 when leaders in various software development methodologies such as Scrum and Extreme Programming held a conference to discuss a way to combat documentation-heavy development processes. Wanting to speed up the development process and empower programmers to mainly be programming instead of writing reports, they created the Agile Manifesto. In the Agile Manifesto, 12 main principles are listed. They are as follows: (1) The highest priority is being able to satisfy the customer through early and continuous software deliverables, (2) be able to change, (3) deliver working software frequently, (4) business people and developers need to work together, (5) trust motivated individuals to get the project done, and give them the tools that they need to succeed, (6) the best way to communicate is with a face-to-face conversation, (7) working software is the best way to measure progress on a project, (8) Agile is about sustainable development; software engineers should maintain a steady pace of development, (9) constantly be aware of good design in the project; this enhances Agile development and makes development easier later on, (10) simplicity is crucial, (11) the best designs and architectures arise from *self-managing teams*, and lastly (12) teams should meet regularly to discuss how they can become more effective [14].

SPECIAL FEATURES

Agile is, at its core, a philosophy and a mindset. Because of this, Agile has shaped many popular development methodologies into what they are today. In addition, Agile is somewhat vague. This allows developers and project managers to take what they want from Agile principles and implement it however they see fit.

GENERAL PROFESSIONAL OPINIONS

Dave Thomas was one of the experts who created the Agile Manifesto. He makes the bold claim that "Agile is dead" [16]. He believes that Agile has been taken over by corporations and made into a more complicated process than it needs to be. However, he still strongly believes that Agile is an important idea and is worth having; it just needs to be reclaimed.

Agile, as Thomas intended it, is a simple framework for how to get work done. Work can be broken up into "what to do" and "how to do it" categories. For the "what to do category", you must first determine where you are in the development process. From there, make small moves toward completing your goal. Adapt your knowledge or way of thinking as a result of what you have learned in the project. Do not continue doing the same things you've always done once you've learned better. Thomas suggests that you repeat this process for as long as you are a software engineer. Thomas also has suggestions for the "how to do it" category. When you have the opportunity to make different choices that will result in roughly the same outcome, choose the development option that will result in making future changes easier. This is akin to smart coding; you add comments in your code so that you have a better idea of what you are doing later; you write clean code. It may take more time to do these things, but it actually saves you more time and effort in the future.

2.4 Scrum

WHAT IS SCRUM?

Scrum is an *iterative* and *incremental* development framework. Much like Agile, Scrum is focused on flexibility with development. However, unlike Agile, Scrum has more definitions and specific development ideas. For example, the Scrum framework outlines the roles employees should have in a Scrum environment. The Scrum team is a small group of people. It usually consists of a *Scrum master*, one product owner, and multiple developers. In the team, Scrum masters are essentially team leads that have special training in Scrum. They take their knowledge of Scrum and teach it to the rest of the team to make development more efficient. They are in charge of monitoring the efficiency of the team regularly. Some Scrum Masters might have some sort of certification. The Scrum.org organization itself offers three different levels of certification: Professional Scrum Master Level I (PSMI), Profressional Scrum Master Level II (PSMII), and Professional Scrum Master Level III (PSMIII). With the PSMI certification, individuals show a "fundamental level of Scrum mastery." For PSMII, there is an "advanced level of Scrum mastery." For PSMIII, the certificate holder has exhibited a "distinguished level of Scrum mastery" [18]. Another role on the Scrum team is the product owner. The product owner manages the *product backlog.* The Product Backlog is essentially the list of *sprints*, or short-development bursts, with specific goals. This backlog keeps track of all of the sprints and their goals for the current project. In order to effectively manage the product backlog, the product owner makes sure that everyone understands the requirement for the upcoming sprints. They also deal with making sure that the product goal gets successfully completed within the Scrum framework.

Scrum is an adaptive framework, but there is a typical Scrum Cycle. First, a product owner takes the complex goals and requirements of a project and puts them into a product backlog. Second, the Scrum team decides together what work from the backlog can be done during one increment. Third, the team takes this decided upon work and turns it into an “Increment of value” [17]. This is an increment of development in which the developers are closer to producing the final product. Fourth, the Scrum team and *stakeholders* review the results. The feedback from the current sprint will be used to adjust the approach to the next sprint. These four steps are repeated until the project is completed.

Scrum is related to Agile in the sense that Scrum applies all of the main principles and goals of Agile. Agile has broad, guiding principles, while Scrum has outlined specific roles in a Scrum Team, whereas Agile leaves the team assignments up to each specific team applying Agile methods themselves. Some of the key elements of Scrum are inspection, transparency, and adaptation. The key Scrum values are commitment, focus, openness, respect, and courage. These elements and values are ultimately very similar to those of 12 key values of Agile.

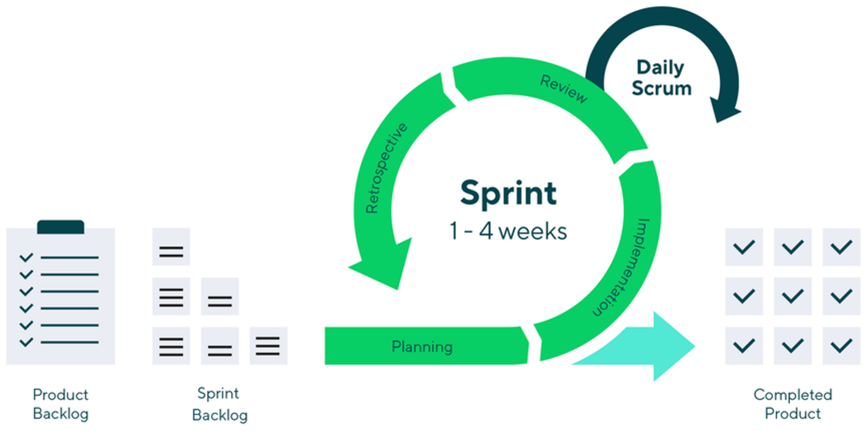

Figure 2.4.1: *The Typical Scrum Sprint* [38]

HISTORY

Scrum was developed in the early 1990s. However, the first Scrum Guide was released in 2010, and additional versions of this guide have been released since then [42]. They all generally hold the same information: definition of Scrum, Scrum Values, Scrum Theory, Scrum Team, Scrum Events, and general definitions of common terms used to describe the Scrum methodology. The Scrum methodology was originally developed for software development, but other fields have been using both Scrum and other Agile-based methods, as these methodologies focus on flexibility and efficiency.

Although the first Scrum Guide was released in 2010, the first public paper about the Scrum framework was published in 1995 by Ken Schwaber. The paper, which is entitled "SCRUM Development Process", was created because of the uncertainty of the software development process [43]. Schwaber was also frustrated with working in the typical corporate environment at that time. Thus, Schwaber developed Scrum. He researched and worked on it for around five years, and his main goal was to develop a software development methodology that would give small teams more control and mitigate the number of unsuccessful projects due to being out of funds or out of time.

SPECIAL FEATURES

Scrum is a lightweight methodology, but there are more “rules” in Scrum than Agile. For example, it gives guidelines for how to go about improving communication: have daily meetings called Daily Scrums. This will help the team get a general idea of where everyone is at in the development process on a daily basis. Scrum also makes recommendations for how to go about improving project efficiency: dividing up work into sprints. There area additional special features, particularly in the form of either Scrum events and roles.

GENERAL PROFESSIONAL OPINIONS

Although Scrum is a popular development framework, there are still complaints. Steven Lowe, a Product Technology Manager at Google, voices these common complaints. According to him, Scrum focuses too much on project planning and meetings [25]. This emphasis on planning can slow down the software development process, which is the opposite of what following a methodology is supposed to do. When a group of developers implement a project management methodology, the main goals are that the quality of work will increase and the time it takes to complete a project will decrease. Lowe also claims that companies often impose Scrum on their employees. He believes that imposing Scrum on individual development teams is anti-Agile. Teams should be able to make their own decisions regarding what will work best for their current projects and according to the team members’ expertise. Furthermore, Lowe believes that companies have ruined Scrum in an attempt to become “Agile”. He states that companies do not really want to invest the proper time and money into making the full transition to Scrum; they simply want the benefits of Scrum and Agile development quickly. This often leads to cutting corners and implementing practices that are not actually endorsed under the Scrum framework.

2.5 Kanban

WHAT IS KANBAN?

Kanban is another Agile-based development methodology: it embraces change. Similar to the other project management methodologies, Kanban’s main goal is to reduce the time it takes to complete a project. The main principles of the Kanban framework can be summarized into three main points. First, one must understand what work they are doing currently. Second, a computing professional should strive to make evolutionary changes throughout the development process. Lastly, if following the Kanban method, workers should be empowered to be leaders in some capacity. These principles guide the common practices of Kanban.

One of the main practices is visualization. This helps a worker understand what they are doing and what they need to do. The Kanban methodology utilizes a *kanban board* to visualize all of the tasks that need to be done in order to complete a project. The kanban board is separated by columns. There is usually a column for new user *stories*, what needs to be done for the sprint, tasks that are currently in progress, tasks that are being tested, and completed user stories.

Another common practice of the Kanban methodology is to limit the amount of work in progress. This means making sure that the development team is focusing on one aspect of the project at a time. In the Kanban methodology, this is a key practice, as it can increase the quality of output and create cohesion in the team unit.

An additional practice of the Kanban methodology is managing the work flow. Managing the flow is determining how often certain people should be working and on what; it also means determining when a certain deliverable should be finished. Teams themselves manage the flow according to what will best help complete the project on time and on budget.

Furthermore, under Kanban, development policies are explicit. Policies should be well-defined. This means that what needs to be done for a project should be readily understood by everyone on the team. It also means that the development process as a whole should be well understood as well. This means that developers should work within the work in progress limits and general development practices that were defined by the team in the past.

It is also a common practice to implement *feedback loops*. Feedback loops are used in Kanban for the team to take time and see what is and is not working so far and make changes accordingly.

In addition, Kanban focuses on improving and evolving software engineering practices incrementally; this means that there are less barriers for any one company to start implementing Kanban. A company can start out with what they are currently doing and making small changes along the way to make their process more Agile. That makes it more obtainable for companies to start following the Kanban methodology sooner.

Jobs in Kanban are kept primarily the same as before. Kanban focuses on taking the existing development method of your team and evolving it as you work on a project. This means that you use the roles currently on your team. If you have a team lead, keep them. If you have a project manager, keep them. Kanban is about adapting your current development plan incrementally to make it Agile.

In the Kanban framework, there is one *single flow* for the whole project. However, there are different times in which feedback loops should ideally be conducted under this methodology. These feedback loops can occur during the Kanban Meeting, Replenishment Meeting, or Risk Review. The Kanban Meeting is simply a daily standup meeting. Developers can discuss in general terms what they have been working on and their current successes or difficulties. The Replenishment Meeting happens on a weekly basis. The team talks about what they have completed and what they will work on next. This is essentially sprint planning, but this planning process for Kanban is less formal. Furthermore, the Risk Review is a monthly meeting in which developers and the team as a whole talk about potential risks or issues with the software itself or the delivery method of the software.

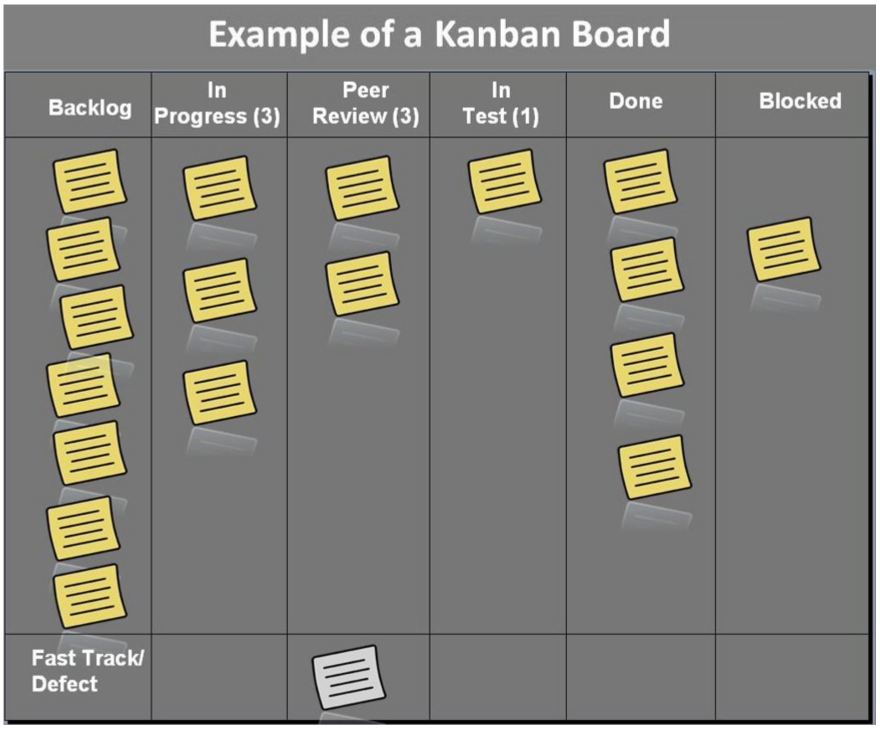

Figure 2.5.1: *Example Kanban Board* [39]

HISTORY

In the Japanese language, "kan" means sign, and "ban" means board. A kanban was a shop sign that communicated what was sold in a clear and concise manner. This became the basis for the Kanban development methodology; it focuses on visual boards called kanban boards that clearly indicates to everyone in the team what has been done and what needs to be done for the project.

Kanban was first used as a production method in Toyota factories in the 1950s. Kanban became more popular in the software development industry after the publication of "The Agile Manifesto" in 2001. Kanban, for the software engineering field, was developed on Agile principles. Kanban boards were first used under the Scrum methodology, but Kanban eventually became its own method. This was because Kanban, which was even more dynamic and flexible than Scrum, helped development teams with aspects of Scrum development that were not efficient. Some of the aspects Kanban improved were ensuring a constant flow of work and shortening the time from getting customer requirements to having a deliverable ready for the customer. Thus, Kanban was seen as a solution to the shortcomings of Scrum.

Similar to other development methodologies, Kanban started gaining more attention after important literature was published. One of these articles was “Kanban vs. Scrum - a practical guide” by Henrik Kniberg [29]. It explored the principles of Kanban, and it was where many developers first learned about the Kanban methodology.

As Kanban grew in popularity from 2010 and beyond, the modern Kanban method was adopted by industries outside of software development. When more companies and industries started to adopt this method, a book called “Essential Kanban Condensed” was published in 2016 by Andy Carmichael and David Anderson [44]. It outlined the goals and principles of Kanban, making it into the development methodology that we know today. These principles made Kanban into a fully-fledged methodology that software developers are following. Without the principles, Kanban is simply a fancy to-do list.

SPECIAL FEATURES

One of Kanban’s most unique features is the emphasis on visualization. Through the use of Kanban boards, developers can see what tasks are listed concisely under each category. Kanban is also easy to start implementing. With Scrum, for example, you would have to restructure your team itself to follow the principles of Scrum. However, the Kanban methodology allows everyone to keep their current roles and make incremental changes to the current process to make it more Agile. In addition, Kanban is highly flexible. This flexibility is akin to what is found in the Agile framework, but Kanban offers more concrete ideas for implementing the methodology, such as using a Kanban board to track what needs to be done. Furthermore, Kanban offers the unique idea of *limiting work*. Kanban focuses on limiting work to a specific feature of a project to ensure rapid development of a particular function and greater team cohesion overall.

GENERAL PROFESSIONAL OPINIONS

Kovair, a software company in Silicon Valley, endorsed an article on their blog that discusses both the advantages and disadvantages of the Kanban methodology. This piece was written by Micael Gorman, a professional academic writer who focuses on writing about technology and project management. He writes that one of the advantages of Kanban is the ease of use [30]. You don't have to receive any sort of certification to begin implementing Kanban correctly. In addition, Kanban is highly flexible. This is key for large projects that are likely going to change over time. Another important advantage of the Kanban framework is its high level of collaboration. With the kanban board, the team can see what needs to be done and see what other people are doing. This promotes collaboration as team members feel like they know what everyone is working on and are able to see how it relates to their task and overall project completion. There are, however, two main disadvantages of the Kanban methodology. The first being that there is a lack of focus. Developers can get distracted because Kanban is so flexible and much more laid back compared to development methodologies like Scrum, in which there is a specific amount of time for each sprint already determined at the start of the sprint. Similarly, there is also a lack of timing. Kanban does not set up specific deadlines as a key part of its methodology. This means it is up to the team to decide when to complete a task or a deliverable for the project.

2.6 Extreme Programming

WHAT IS EXTREME PROGRAMMING?

Extreme Programming, commonly referred to as XP, is an Agile software development framework. It is considered the most specific Agile framework. It specifically outlines what practices you should be following in order to be accurately abiding by the XP framework. XP also details the values one should have if following the framework, with the first value being communication. Communication is important, but some forms of communication are better than others. According to the XP framework, the best type of communication is face-to-face with a whiteboard nearby for visualizing problems. Another value is simplicity. To apply this value, you should keep the design of the system as simple as possible. You should also avoid waste, and only do absolutely necessary things during development. Thus, you should only think about the project requirements that you know about; do not worry about potential requirements that could arise in the future. The third key value is feedback. Under Extreme Programming, constant feedback is important. It helps teams realize what they are doing right and what they are doing wrong. The fourth is somewhat unconventional; it is courage. We typically do not associate programming with courage, but XP places much importance on it. In the XP framework, having courage is accepting and acting on difficult feedback. You also need to have courage to tell your team what you think is going poorly in the development process. The last value is respect. This is fairly simple: respect everyone on your team, and acknowledge that respect produces better communication.

There are numerous official practices that one must follow in order to accurately be applying the XP methodology. First, if possible, team members should sit near each other. Face-to-face communication is important while working on a project, and sitting physically nearby makes it easier to communicate often. If you are following XP, you must also enforce *pair programming.* Pair programming is software development by two people at the same machine. Another practice is the use of stories. Stories describe what the final product should be able to do in terms of what various users want. Those who claim to be following XP should also adhere to a weekly development cycle, and the team can pick a day to meet each week and reflect on progress. Interestingly enough, the XP methodology wants to take into consideration slack. Teams should account for some low priority tasks or stories in development cycles that can be dropped in favor of completing more important tasks. *Continuous integration* is another key practice. This means that code is merged often and is always tested right away when it is added to a larger portion of code. XP also promotes the practice of *test-driven development.* The normal basic development path is developing the code, writing tests, running tests, and repeating the process. The test-first programming flow is first writing failing automated tests, running the failing tests, developing code that passes the tests, running the tests, and repeating the process. XP also is in favor of incremental design. This means that certain features should be developed at a specific time; the whole project should not be worked on all at once.

XP specifies job roles that people in the organization should have. The official roles under the Extreme Programming framework are as follows: customer, developer, *tracker*, and *coach.* The customer makes business decisions regarding the project. The developers realize the stories and work on the project. The tracker is an optional role. This person would keep track of metrics or information that tracks progress, successes, and areas for improvements. This role is akin to the traditional job of a project manager. The coach role is usually someone who is part of other development teams in the company. It is someone who has used XP before and can act as a mentor when it comes to XP practices.

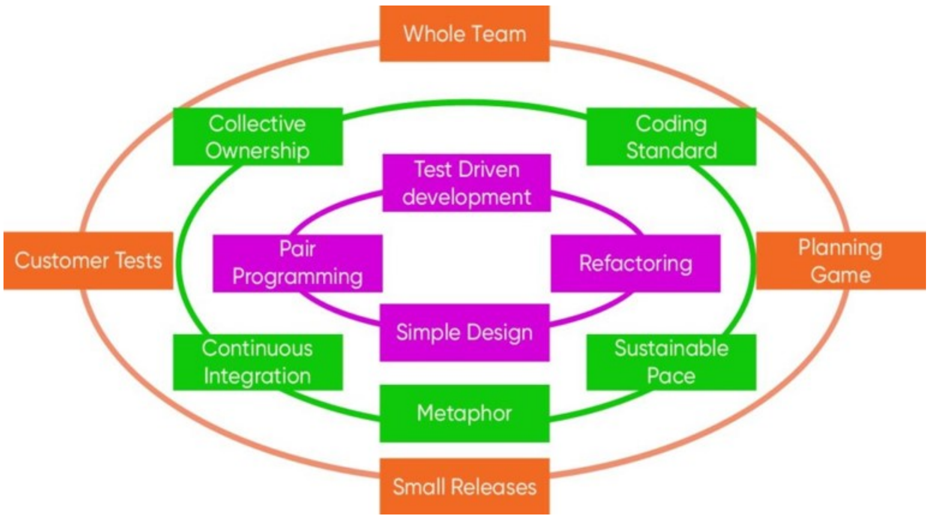

Figure 2.6.1: *Main Principles behind XP* [40]

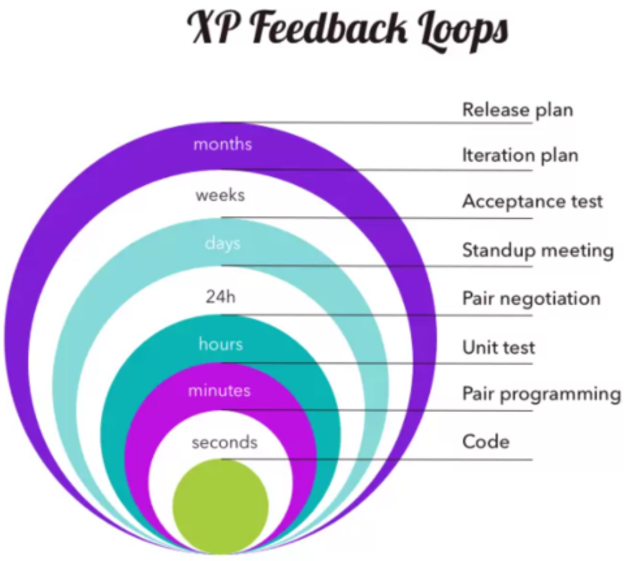

Figure 2.6.2: *XP's Feedback Loop* [41]

HISTORY

Kent Black can be seen as the original thinker behind Extreme Programming. Black was working for Daimler-Chrysler, an automotive manufacturing group, when he came up with the basic tenets of XP: improve communication, simplicity is best; everyone needs feedback, and you need to have courage. Much like the other Agile-based methods, XP focuses on being able to adapt to constant change. XP takes the typically simple, ideal practices of software development and pushes it to an extreme. For example, XP requires code reviews, pair programming, unit testing, and even has the customer test the functionality themselves.

SPECIAL FEATURES

XP gives detailed instructions on how to apply its principles. It guides developers on where to sit, how to communicate, when to test, when to meet as a team, and how to program. This is different from the general Agile framework, as Agile is fairly open to interpretation. As long as you embrace change and are adaptive, you are technically following Agile. However, to follow XP, you need to follow the principles as well as the actual practices of pair programming, like doing a weekly review and sitting by your team members. XP also places emphasis on getting continuous feedback from the customer and testing. Testing is important. Under the XP framework, even the customer tests the product. The flow of programming under XP is unique as well. It follows a test-first programming pattern, where one would write a failing automated test, run that failing test, develop code that passes the test, re-run the test, and repeat that process. This test-driven development makes XP very focused on the outcome of the project as a whole, as you want to be able to address issues and pass the tests as quickly as possible.

GENERAL PROFESSIONAL OPINIONS

There are of course benefits and detriments to XP. The pros are that there is faster development, open communication, and enhanced teamwork. There are also strong cons. It is difficult to follow XP when the client is not located near the development team, as one of the principles is that communication is best when done face to face. There is also a lack of documentation because of constant changes and stress due to tight deadlines, which is why it is important to include slack.

Many resources also talk about the intense customer involvement required in XP as a potential issue. It is unrealistic and burdensome to have a customer involved in every single development change made. There could also be an issue with communication. Technical individuals and business workers usually speak a different language, and these workers have jobs outside of communicating with one another. Furthermore, an on-site customer is expensive, and the customer may not have time to meet with the development team that often. The customer often has other projects or different responsibilities.

3 Methodology Comparison

After considering the general features and aspects of the six methodologies in the previous section, this next section will explore the more in-depth study of the similarities and differences between the methodologies. This section's scope includes a discussion of the different development environments in which each methodology ideally thrives. A development environment can be comprised of various aspects, but in this context, a development environment will particularly cover the types of projects being worked on and the types of team members working on the project. In addition, this section will discuss the general advantages and disadvantages of each framework.

3.1 Ideal Development Environment

Certain projects or development environments are better suited for specific development methodologies. All Agile-based methods, such as Extreme Programming, Kanban, Scrum, and Agile itself, thrive in projects that are expected to be complex and long-lasting. They also are well-suited for changing environments and projects. These Agile methods address change and flexibility; thus, working on a project that will change is perfectly suited for these methodologies.

Surprisingly, there are similarities found between Extreme Programming and the Spiral Model. Although the Spiral Model is a software development life cycle, it focuses on addressing potential project risks and minimizing those risks. Extreme Programming also focuses on project risks, especially those projects that may have issues with the team utilizing foreign technologies. Thus, both XP and the Spiral Model are good for addressing projects that have high-cost risks associated with them.

In addition to being ideal for projects that are changing, two frameworks are similar in the way that they promote the development and delivery of executables. These methodologies are the Kanban method and the Spiral Model. With the Spiral Model, deliverables are often in the form of prototypes. In the Spiral Model, during every round of development, a new prototype is finished and shown to the respective clients and stakeholders. Kanban also focuses on completing deliverables, as Kanban champions the idea of limiting work in progress. This means that teams can hone in on a specific functionality at a time, which indicates that there are more small-scale deliverables ready in less time.

The main differences in the ideal project for different methodologies are primarily found in the differences between the Agile-based methods and the Waterfall Model. The Waterfall Model is best for short projects that are not considered as complex. This Waterfall method is also good when the requirements are clear, well-documented, and unlikely to change. The Waterfall Model can also be used on a team that does not have experience utilizing Agile methods, as learning how to implement the more complex Agile-based frameworks takes practice and dedication. If you have to learn Agile methods, there is also considerable time taken away from development. Agile-based methods, however, are the champions of long and complex projects that are often subject to change. This is because the flow of work each Agile method promotes usually involves some time to meet with the development team and adjust the development plan. Through meeting frequently to adjust the development plan and acknowledging beforehand that change is imminent, these Agile methodologies are the best in industries and projects with changing requirements.

3.2 Advantages and Disadvantages

Each project management methodology comes with its own set of advantages and disadvantages. For example, Agile's main advantage is that it does not attempt to predict the future. It accepts change and uncertainty, and this gives way to flexibility. There is also less work in the beginning phases, as there is a lack of formal documentation that is required under the Agile methodology. This also means that the team can start working on the project faster. However, even a framework as popular as Agile has its downsides. There is generally a lack of understanding when one reads the official Agile literature. What does it actually mean to be Agile? It is very vague, and it is essentially just a set of principles that one should use to guide their development process. Due to this, many companies do not implement Agile correctly. They simply want to reap the benefits, but they do not want to make the investments in time, effort, and money to actually be an Agile corporation. This is mainly due to the fact that Agile requires extreme flexibility and familiarity with uncertainty, and most businesses are uncomfortable with a lack of deadlines and documentation.

Unlike Agile, Scrum provides a fairly clear plan for teams to follow. The roles, such as Scrum master, product owner, and developer, are well defined. Reviewing and planning also takes place often under the Scrum methodology, which makes it easier to mitigate design problems earlier in the development process. Having the specific roles and planning processes to follow are often seen as one of Scrum's advantages in the software engineering environment. But there are of course disadvantages to following such a rigid plan. Scrum focuses so much on project planning and meetings that it could potentially slow down development if not done properly. Scrum is also considered to be more complex than Agile, with the specific roles, meetings, and sprints.

While Scrum is known for its complexity, Kanban is actually known for its ease of use. With Kanban, everyone can keep the same job titles and general responsibilities. Kanban also has a unique advantage in that it uses a kanban board; this promotes collaboration, as the whole team can see what others are working on or have already completed. In addition, the kanban board itself involves a simple, visually-appealing format that is often liked by developers because they are uncomplicated to implement and easy to comprehend. Kanban's ease of use, however, means that there is a lack of focus in this development. This is because Kanban is a very flexible methodology, which means that there ideally should not be any set times for sprint completion. There should only be guidelines and ideals for completion dates.

Extreme Programming's main strength lies in its development speed. Extreme Programming (XP) champions the practice of test-driven development. By following test-driven development, projects can ideally be expected to be completed at a faster speed than if linear development was used. This is because writing tests first can help reveal problems in the software more quickly than code without test cases, and it also gives developers the ability to test isolated pieces of code. Furthermore, another advantage of following XP is the emphasis it places on open communication. XP makes it clear that communication should happen frequently during the development process. Although communication is crucial to any software development project, this communication could potentially be seen as one of XP's downsides. XP promotes face-to-face communication, and with more developers working remotely, that may be difficult for teams to entirely follow. In addition, XP is intensely involved with the customer, and this is ideal to some extent, but the customer and stakeholders often have other projects or different responsibilities to attend to as well. This means that they often cannot meet with the development team as much as the XP framework suggests to do.

Now considering the SDLCs, which are entirely different from the Agile-based methods, we can see that they have their own challenges and advantages as well. For the Waterfall Model in particular, this methodology is very simplistic. There are six simple steps that one must follow. This structured process is easy to follow and places an emphasis on creating documentation. Because the Waterfall Model is so simplistic, there are many missing components to this methodology. For example, there is no feedback step or path in this model. This means that there is not any formal time for developers to reach out to stakeholders to get feedback. In addition, there is a strict policy of not overlapping phases. This can be unrealistic in the field, where functions often have to be worked on at the same time to promote efficient development. Another main disadvantage of following the Waterfall Model is the lack of acknowledgement of the uncertainty and change that are found within software development. The Agile methods embrace change, making them more flexible, while one has to theoretically run through the entirety of the Waterfall Method again in order to address change.

The Spiral Model is often seen as a more advanced, upgraded version of the Waterfall Model. This is perhaps because they are both SDLCs. The Spiral Model places emphasis on risk analysis; this is a major benefit, as this can help prevent major issues in the development process. Another key benefit of adhering to the Spiral Model is that prototypes are made often. These prototypes can be used to ascertain how well the current project progress is coming along, and these prototypes also enable teams to get feedback from stakeholders. However, the number of prototypes that are created as a result of following the Spiral Model make this an expensive methodology. In addition, the time spent planning and doing risk analyses could be considered excessive and take away time from programming.

4 Conclusion

The focus of this paper is on the Waterfall Model, Spiral Model, Agile, Scrum, Kanban, and Extreme Programming methodologies. These methods were discussed in a general manner in order to promote better understanding of what the formal documentation says about how to follow these development frameworks. The same aspects of each methodology were also discussed. For example, the general practices, roles, and values are considered in the first part of the paper. The history, professional opinions, and unique features are also considered. The methodologies are also compared to one another in another section of the thesis, particularly the best environments and the advantages and disadvantages relative to other methods.

Within the paper, it was determined that one of the key features championed by Agile and the similar methodologies of Scrum, Kanban, and Extreme Programming is flexibility. Software engineering projects are often bound to change. There are elements of development that cannot be controlled. If we acknowledge that change is going to happen unexpectedly, it can relieve stress and promote incorporating slack into a project's general development plan. Because of embracing change, Agile-based methods are more likely to be used for larger, more complex projects. Still, the Spiral Model and the Waterfall Model have their place within the world of software development. They are more ideal for shorter-term projects in which the requirements are well-defined and unlikely to change. However, it should be mentioned that much of the research in this paper is more theoretical in nature, considering that much of the benefits of following a method arises from following it exactly as it is formally documented; all the team roles, meetings, and development processes must be exactly the same on paper as they are in real life. This is entirely unrealistic. People can unintentionally implement an un-Agile practice when they still say the are prescribing to Agile as a whole. Thus, we cannot expect software engineers and project managers to always follow their prescribed methodologies exactly. Because of this, there was an attempt to include more real-life problems and experiences by including varying professional opinions of each methodology. These professionals take into consideration how each methodology is actually implemented within an organization, not how a methodology should ideally be implemented.

The research in this paper was presented with the hope of fostering a general understanding of common software engineering methodologies. If software engineering methodologies are better understood, development could be made much more efficient. This saves companies time and money, and it also saves developers from immense frustration. Additionally, if software engineers focused more on how they should go about developing their software, they may feel a greater sense of agency over their own projects and feel as if they have a clearer path as to what needs to be developed at a particular time. Therefore, by focusing on general aspects of various development frameworks, this paper strives to foster more interest in development methodologies. If implemented properly, these methods can be of great importance in the software engineering world, as mentioned, potentially saving two important resources: time and money. If implemented incorrectly, software methodologies can actually make one's work flow more confusing and more inefficient. Thus, researching different methodologies is a worthwhile endeavor, as it can hold the difference between a successful project and an unsuccessful project.

With additional time, the scope of this study could be extended to create practical software that could help individuals who are unsure of which development methodology they should be using. This software could take the form of a well-designed survey in which software engineers could input information about their organization, team, project, and personal opinions. Different answers will have different point values associated with them. At the conclusion of the survey, the points will be totaled, and they will fall within a particular range. The range will correlate to an ideal software development methodology for the information given by the user. The result will then be presented to the survey taker, who can consider using that methodology for their project's development.

5 Glossary

These terms are found throughout the paper in italicized text. When relevant, the vocabulary is explained in the paper, but there are occasions when they are not. Thus, this glossary will ease any uncertainty about the definition of a term within the context of the thesis.

5.1 Terminology

Definition 5.2.1. **Acceptance testing** is a form of testing done by a client. The client will decide either to accept or reject the software based on how the developers dealt with the the requirements given to them.

Definition 5.2.2. **Adaptive maintenance** is a type of maintenance done to preexisting software when it needs to be moved to a new platform or host environment.

Definition 5.2.3. The term **Agile-based methods** refers to the methods that are similar to Agile. This ultimately means that these methods are flexible and adaptive in nature.

Definition 5.2.4. In Extreme Programming, the **coach** role is someone who has experience working with the Extreme Programming methodology. The coach can guide a team new to Extreme Programming to ensure that they are implementing it properly and experiencing its full benefits.

Definition 5.2.5. **Continuous integration** is a type of integration in which every individual's code is merged with the larger code base of the software project through an automated, regularly-scheduled process.

Definition 5.2.6. **Corrective maintenance** is a form of maintenance done on released software. This maintenance occurs when errors that were found in the software after it went live need to be corrected. After corrective maintenance is performed on the software, the software is released once more.

Definition 5.2.7. **Deliverables** are various updates, goods, or services that can be "delivered" to the customer. Some examples of deliverables are reports, documents, or working pieces of software.

Definition 5.2.8. The term **developers** refers to the individuals who work on creating, updating, and maintaining the software in a project.

Definition 5.2.9. There are different types of **documentation** in the software engineering field. Documentation catalogues the ideas and progress associated with a project. Different methodologies require varying levels of documentation, which can take the form of reports or informal notes.

Definition 5.2.10. **Executables** are pieces of working software that are delivered to the client to get approval.

Definition 5.2.11. In Kanban, **feedback loops** encompass the idea that feedback will be given continuously throughout a project, and the feedback that is given will be acted upon in some manner.

Definition 5.2.12. **Incremental** development is the process of building off of the work done in previous tasks.

Definition 5.2.13. **Integration testing** is a type of key software testing in which individual parts are combined to test the functionality of the entire project.

Definition 5.2.14. **Iterative** development is a type of development in which the larger pool of work is split into smaller parts to be completed separately.

Definition 5.2.15. A **kanban board** is the main feature of the Kanban methodology. A kanban board is used to visualize the different phases of a task or story. These items may potentially be in the columns that represent the work being new, in progress, tested, or complete.

Definition 5.2.16. Kanban offers the idea of **limiting work**. A team may focus on specific functionalities or features of a software at a time instead of completing the entire project at once.

Definition 5.2.17. A **linear-sequential life cycle model** is a form of development that follows a particular, direct path of development consistently every time the model is used.

Definition 5.2.18. **Pair programming** is a practice endorsed by Extreme Programming. Pair programming is when two engineers develop software together at the same time on one machine.

Definition 5.2.19. **Perfective maintenance** is a form of maintenance done to live software in an effort to enhance existing features of the software.

Definition 5.2.20. A **product backlog** is a concept found in Scrum. A backlog is a essentially a list of work that the team needs to complete.

Definition 5.2.21. In Scrum, a **product owner** is the person who manages the product backlog. This person additionally manages the team's progress as well. Thus, the product owner is akin to a project manager.

Definition 5.2.22. A **prototype** is a model or portion of software before it is in its finalized form.

Definition 5.2.23. A **retrospective** is a meeting held at the end of a project or period of time to reflect what was successful and unsuccessful in the development process.

Definition 5.2.24. **Risks** are any stumbling block that can cause a project's development to take too long or cost too much.

Definition 5.2.25. In Scrum, a **Scrum master** is a person who works closely with the development team. They serve as a team leader, guide the development of a project, and provide resources to the developers.

Definition 5.2.26. **Self-managing teams** are teams that have the power to determine who completes what task and what software development methodology to follow without the interference of an external member of the organization or company.

Definition 5.2.27. A **single flow** is one development pace that is kept consistent throughout a determined period of time.

Definition 5.2.28. **Slack** is the idea that lower-priority features could be worked at a later date. They would ideally be completed by a certain time, but this completion is not required. This relieves pressure from the software engineers if the higher-priority items take longer than expected to develop.

Definition 5.2.29. A **software development life cycle model (SDLC)** is a general development model in which the steps for work working on software are well-defined and occur in a particular order.

Definition 5.2.30. A **software requirement specification (SRS) document** is formal documentation that outlines the expected functionalities of a software that is to be completed for a client. If an SRS document is needed for a project, it usually happens that the client will have to agree on the information outlined in the document before development proceeds.

Definition 5.2.31. **Sprints** are a predetermined length of time in which outlined development tasks are to be completed.

Definition 5.2.32. **Stakeholders** are all individuals who have an interest or claim in the success of a project that is being developed.

Definition 5.2.33. **Stories** are development features or tasks.

Definition 5.2.34. A **system** refers to software developed in a project that works together.

Definition 5.2.35. **Test-driven development** is a process in which failing tests are written first before any functionality is developed. After the tests have been created, code is written to pass the tests.

Definition 5.2.36. In Extreme Programming, a **tracker** is an optional team role. A tracker would keep track of a team's progress and provide them with metrics that show the efficiency of their development. A tracker is similar to a traditional project manager.

Definition 5.2.37. **Unit testing** is testing done on a specific, small portion of the code to test stand-alone functionality.

6 References

[1] SDLC - Waterfall Model. (n.d.). Retrieved from `https://www.tutorialspoint.com/sdlc/sdlc_waterfall_model.htm`.

[2] GeeksforGeeks. (2021, October 21). *Software Engineering: Classical Waterfall Model.* GeeksforGeeks. Retrieved from
`https://www.geeksforgeeks.org/software-engineering-classical-waterfall-model/`.

[3] Hughey, D. (2009). *Comparing Traditional Systems Analysis and Design with Agile Methodologies.* The Traditional Waterfall Approach. Retrieved from `https://www.umsl.edu/~hugheyd/is6840/waterfall.html`.

[4] *SDLC - Overview.* tutorialspoint. (n.d.). Retrieved from `https://www.tutorialspoint.com/sdlc/sdlc_overview.htm`.

[5] GeeksforGeeks. (2020, May 10). *Advantages and Disadvantages of using Spiral Model.* GeeksforGeeks. Retrieved from
`https://www.geeksforgeeks.org/advantages-and-disadvantages-of-using-spiral-model/`.

[6] IEEE. (n.d.). *Barry Boehm.* IEEE Computer Society. Retrieved from `https://www.computer.org/profiles/barry-boehm/`.

[7] B. W. Boehm, "A Spiral Model of Software Development and Enhancement," in *Computer*, vol. 21, no. 5, pp. 61-72, May 1988, doi: 10.1109/2.59.

[8] Gershick, Z. Z. (1994, January 17). *Barry Boehm Named TRW Professor in Software Engineering.* USC News. Retrieved from
`https://news.usc.edu/7503/BARRY-BOEHM-NAMED-TRW-PROFESSOR-IN-SOFTWARE-ENGINEERING/`.

[9] Cooke, J. L. (2016). *Agile - An Executive Guide: Real results from IT budgets.* It Governance Pub.

[10] Agile Alliance. (n.d.). *Agile 101.* Agile Alliance. Retrieved from `https://www.agilealliance.org/agile101/`.

[11] Visual Paradigm. (n.d.). *Agile Development: Iterative and Incremental.* Visual Paradigm. Retrieved from
`https://www.visual-paradigm.com/scrum/agile-development-iterative-and-incremental/`.

[12] Highsmith, J. (2001). *History: The Agile Manifesto.* Agile Manifesto. Retrieved from `https://agilemanifesto.org/history.html`.

[13] Rigby, D., Sutherland, J., Takeuchi, H. (2016, April 20). *The Secret History of Agile Innovation.* Harvard Business Review. Retrieved from `https://hbr.org/2016/04/the-secret-history-of-agile-innovation`.

[14] Agile Manifesto. (2001). *Principles Behind the Agile Manifesto.* Agile Manifesto. Retrieved from `https://agilemanifesto.org/principles.html`.

[15] Haworth, S. (2021, January 15). *Agile Vs Waterfall: When To Use How To*

Implement Hybrids. The Digital Project Manager. Retrieved from `https://thedigitalprojectmanager.com/agile-vs-waterfall/#2-which-methodology-use`.

[16] GOTO Conferences. (2015, July 14). *Agile is Dead • Pragmatic Dave Thomas • GOTO 2015* [Video]. YouTube. Retrieved from `https://www.youtube.com/watch?v=a-BOSpxYJ9M`.

[17] Scrum.org. (n.d.). *What is Scrum?* Scrum.org. Retrieved from `https://www.scrum.org/resources/what-is-scrum`.

[18] Scrum.org. (n.d.). *What is a Scrum Master?* Scrum.org. Retrieved from `https://www.scrum.org/resources/what-is-a-scrum-master`.

[19] Scrum.org. (n.d.). *What is a Product Owner?* Scrum.org. Retrieved from `https://www.scrum.org/resources/what-is-a-product-owner`.

[20] Scrum.org. (n.d.). *What is a Product Backlog?* Scrum.org. Retrieved from `https://www.scrum.org/resources/what-is-a-product-backlog`.

[21] Scrum Guides. (n.d.). *The 2020 Scrum Guide*. Scrum Guide — Scrum Guides. Retrieved from `https://scrumguides.org/scrum-guide.html`.

[22] Lobellova, V. (2020, January 22). *The History of Scrum: How, when and why*. ScrumDesk, Meaningfully Agile. Retrieved from `https://www.scrumdesk.com/the-history-of-scrum-how-when-and-why/`.

[23] Wood, M. (2013). *Why You're Confusing Frameworks with Methodologies*. Project Management. Retrieved from `https://www.projectmanagement.com/articles/278600/why-you-re-confusing-frameworks-with-methodologies`.

[24] Srivastava, A., Bhardwaj, S., Saraswat, S. (2017). Scrum model for Agile methodology. *2017 International Conference on Computing, Communication and Automation (ICCCA)*. `https://doi.org/10.1109/ccaa.2017.8229928`

[25] Lowe, S. A. (2019, June 26). *Why Scrum sucks*. TechBeacon. Retrieved 2021, from `https://techbeacon.com/app-dev-testing/why-scrum-sucks`.

[26] Agile Alliance. (2021, March 4). *What is Kanban?* Agile Alliance. Retrieved 2021, from
`https://www.agilealliance.org/glossary/kanban/#q=~(infinite~false~filters-\protect\@normalcr\relax(postType~(~'page~'post~'aa_book~'aa_event_session~'aa_experience_report~'aa_glossary~'aa_research_paper~'aa_video)~tags~(~'kanban))~searchTerm~'~sort~false~sortDirection~'asc~page~1)`.

[27] Rehkopf, M. (n.d.). *Kanban vs Scrum*. Atlassian. Retrieved from `https://www.atlassian.com/agile/kanban/kanban-vs-scrum`.

[28] Kukhnavets, P. (2021, September 28). *What is Kanban Board: Tasks, Workflows, Processes*. Hygger.io. Retrieved from `https://hygger.io/guides/agile/kanban/kanban-boards/`.

[29] Kanban Tool. (2021, November 8). *History of Kanban. Kanban Tool.* Retrieved from https://kanbantool.com/kanban-guide/kanban-history.

[30] Gorman, M. (2020, December 22). *Scrum VS Kanban: Weighing Their Pros and Cons.* Kovair Blog. Retrieved from https://www.kovair.com/blog/scrum-vs-kanban-pros-and-cons/.

[31] Agile Alliance. (2021, March 10). *What is Extreme Programming (XP)?* Agile Alliance. Retrieved from https://www.agilealliance.org/glossary/xp/#q=~(infinite\protect\@normalcr\relax~false~filters~(postType~(~'post~'aa_book~'aa_event_session~'aa_experience_report~'aa_glossary~'aa_research_paper~'aa_video)~tags~(~'xp))~searchTerm~'~sort~false~sortDirection~'asc~page~1).

[32] Hutagalung, W. (2006). *Extreme Programming.* Retrieved 2021, from https://www.umsl.edu/~sauterv/analysis/f06Papers/Hutagalung/.

[33] Buencamino, P. (2013). *Extreme Programming.* Retrieved 2021, from https://www.umsl.edu/~sauterv/analysis/Fall2013Papers/Buencamino/Extreme%20Programming/ExtremeProgramming.html.

[34] Panayotova, E. (2018, December 24). *What Are the Pros and Cons of Extreme Programming (XP)?* Simple Programmer. Retrieved 2021, from https://simpleprogrammer.com/pros-cons-extreme-programming-xp/.

[35] Royce, W. (1970). *Managing the Development of Large Software Systems.* Retreived 2021, from http://www-scf.usc.edu/~csci201/lectures/Lecture11/royce1970.pdf.

[36] Image Courtesy of *GeeksforGeeks*, from https://www.geeksforgeeks.org/advantages-and\-disadvantages-of-using-spiral-model/.

[37] Image Courtesy of *Project Management*, from https://project-management.com/agile-manifesto/.

[38] Image Courtesy of *Wrike*, from https://www.wrike.com/scrum-guide/scrum-sprints/.

[39] Image Courtesy of *Wikipedia*, from https://en.wikipedia.org/wiki/Kanban_(development).

[40] Image Courtesy of *agility.im*, from https://agility.im/frequent-agile-question/what-is-extreme-programming/.

[41] Image Courtesy of *digite*, from https://www.digite.com/agile/extreme-programming-xp/.

[42] Scrum Alliance. (2010). *Scrum Guide.* Retrieved 2021, from https://res.cloudinary.com/mitchlacey/image/upload/v1589750495/Scrum_Guide_v1_Scrum_Alliance_qe0y2w.pdf

[43] Schwaber, K. (1995). *SCRUM Development Process.* Retrieved 2021, from http://www.jeffsutherland.org/oopsla/schwapub.pdf.

[44] Anderson, D., Carmichael, A. (2016). *Essential Kanban Condensed.* Retrieved

2021, from https://dl.acm.org/doi/10.5555/3052276.