Agile Software Development Approaches and Their History

Volkan Günal

August 3, 2012

ABSTRACT

Author: Günal, Volkan

Enterprise Software Engineering 2012: Agile Software Development (Seminar)

With the constant development of information systems and as the business has taken different forms over the years, a number of the traditional methods have turned into highly documentation-oriented, heavy ways of development and such models strictly requiring and limiting developers to follow and apply certain and irreversible processes.

As a counter-reaction to those problems the traditional methods had brought, the understanding of:

- the significance of human factor,

- importance of collaboration and communication between the team and the customers,

- and the value of ability to respond to changes

has begun to arise in the software industry. This leaded the new Agile Methods be formulated and applied over the last two decades, in order to overcome the shortcomings of the traditional methods.

This paper aims to analyze the concepts lying behind Agile Software Development approaches, considering *The Agile Manifesto* and its values as the base to follow.

Describing the values of The Agile Manifesto and focusing on some of the most commonly used agile techniques, such as eXtreme Programming, SCRUM et al.; provides a clear view to see the key points and the advantages of Agile Software Development and understand why it is used over the traditional methods, due to its collaborative, communicative, flexible, fast and efficient properties.

Chapter 1

Agile Software Development Approaches and Their History

1.1 Introduction

Volkan Günal

Software plays a very crucial role in numerous number of areas of technology and business world, as it is driven extensively both by the individuals and the companies either as a single main application or as a part of an aggregate project in order to ease the level of effort, raise functionality and consistency of the work by computerizing procedures and enabling services.

Due to the constant development of information systems, the increasing demand in the field and as the business has taken different forms over the years, different software development methods and models have been invented and used over the last five decades in order to facilitate the development processes.

1.1.1 Realization

With the aforementioned qualities, some of the traditional methods have turned into highly documentation-oriented ways of development and such models strictly requiring and limiting developers to follow and apply certain processes. As a counter-reaction to those problems the traditional methods had brought, the understanding of the significance of human factor, importance of collaboration and communication between the team and the customers, and the value of ability to respond to changes has begun to arise in the software industry. This leaded the new Agile Methods be formulated and applied over the last two decades, in order to overcome the shortcomings of the traditional methods.

Having an initial look at Toyota's manufacturing and production philosophy considering these values helps us understand the concept of "continuous change" and "human orientation" leading to success, before focusing on the Agile Software Development.

1.1.2 Overview of Toyota Way 2001

Toyota Motor is a global automotive corporation that has a well-earned reputation for excellence in quality, cost reduction, and hitting the market with vehicles that sell.[14, p. 3] Toyota is a good example of a company which applied basics of agile principles in a production-oriented environment since 1980s.

In April 2001, Toyota adopted the *Toyota Way 2001*, an expression of the values and conduct guidelines that all employees should embrace in order to perform the guiding principles of the corporation successfully.[1, p. 80] In fact, its roots lie on the summer training of supervisor's in 1987, where the system was fully articulated by then, as was basic company philosophy, especially in such areas as quality and human resources. [14, p. xvi] The philosophy behind the *Toyota Way 2001* is explained in two concepts as *Continuous Improvement* and *Respect for People* in the company's

Environmental and Social Report 2003, as seen in Figure 1.1 .

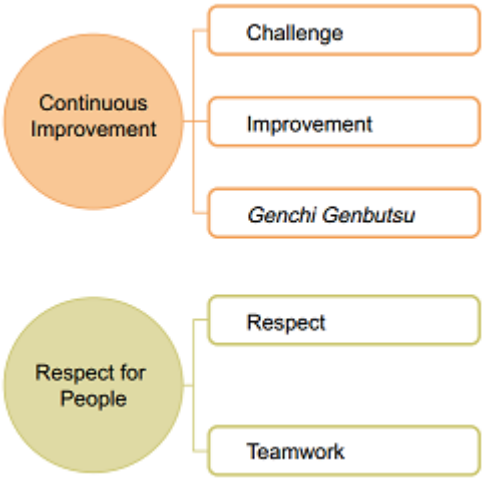

Figure 1.1: Key Principles of Toyota Way [1]

In 2004, Jeffrey Liker published his book "The Toyota Way", in order to formulate the *Toyota Way 2001* and provide a guide to succeed in manufacturing based on the principles he defined. Liker expresses the idea that *Toyota Way* is a system designed to provide the tools for people to continually improve their work, meaning more dependence on people. [15, p.36]

Liker summarizes the principles of Toyota Way in four separate sections, namely as:

1. *Long-Term Philosophy - as management decisions on a long-term philosophy, even at the expense of short-term financial goals*

2. *The Right Process Will Produce the Right Results*

3. *Add Value to the Organization by Developing Your People*

4. *Continuously Solving Root Problems Drives* [15, p. 37-40]

1.1.3 Structure and Research Topics

This paper aims to analyze the concepts lying behind Agile Software Development approaches, considering The Agile Manifesto and its values as the base to follow.

Therefore, in Section 1.2 the four main values of *The Agile Manifesto* are described with its ideological background, also by reminding the twelve principles of being agile.

In Section 1.3 , focusing on some of the most commonly used agile techniques, such as eXtreme Programming, SCRUM, Adaptive Software Development, Crystal et al., provides a clear view to see the key points and the advantages of Agile Software Development and understand why it is used.

In Section 1.4 , a comparison is made between the Agile Software Development and the traditional methods, such as Waterfall approach, to see the differences and how Agile Methods aims to overcome the shortcomings of the traditional methods.

1.2 Agile Software Development

1.2.1 Agility

To be able to fully understand the concept, it is helpful to first describe the meaning of the term "*agility*" from different perspectives.

Back in 1995, the term agility was defined well in *Agile Competitors and Virtual Organizations*, in fact for flexible manufacturing, as "*Agility is dynamic, context-specific, aggressively change-embracing, and growth-oriented. It is not about improving efficiency, cutting costs, or battening down the business hatches to ride out fearsome competitive "storms."It is about succeeding and about winning: about succeeding in emerging competitive arenas, and about winning profits, market share, and customers in the very center of the competitive storms many com-*

panies now fear." [10], which is such a description that still keeps its validity today.

According to Highsmith (2002), the most clearly focused definition of agility is that it is the ability to both create and respond to change with the purpose of profit in a turbulent business environment. [12, p. 16]

Furthermore, one of *The Agile Manifesto*'s authors, namely Alistair Cockburn (2001), emphasizes that the use of light-but-sufficient rules of project behavior and the use of human- and communication-oriented rules is core to agile software development [7, p. 8], considering the agility on software development.

1.2.2 The Agile Manifesto

In February of 2001, seventeen practitioners of several programming methodologies came together at a summit in Utah to discuss the problems of existing methodologies, the ways to overcome those, and the values to support agile or lightweight software development at high level; then they published *The Agile Manifesto* with the four main values that were agreed on as:

*"**Individuals and interactions** over processes and tools*

***Working software** over comprehensive documentation*

***Customer collaboration** over contract negotiation*

***Responding to change** over following a plan"* [4]

With these four main values, *The Agile Manifesto* simply and clearly states what is more important for a better software development. The formulation of *The Agile Manifesto* in more detail helps us comprehend and follow the further elaboration of Agile Software Development, and distinguish the traditional methods from it.

Individuals and Interactions over Processes and Tools

This first asset of *The Agile Manifesto* aims to express that the people are more significant than certain processes and tools. Most of the traditional methods, such as Capability Maturity Model (CMM), Waterfall approach et al., have processes that are based on roles. And, this basically implies that the humans are replaceable; which is yet mostly an inconvenient case for developing a software where individuality is crucial. Hence, replacement and/or substitution of individuals is not easy in software development.

This value also points out the fact that individual interactions are core to the discussions between people to obtain new solutions, thus the quality of the interactions matter, as Cockburn (2001) defends this idea by stating that he rather prefers an undocumented process with good interactions than a documented process with hostile interactions. [7, p. 178]

Working Software over Comprehensive Documentation

Documentation (of the requirements, designs, sequence charts, flows et al.) is a useful part of a software development process, since it helps us visualize concepts, specify and follow the requirements, gather information and observe unreliable and/or reliable measurements. Consequently, this value is not strictly opposed to documentation; yet it expresses that the documents should be used as markers[7, p. 179] , in other words as a way to draw the future of the project by hints. Hence, rather than a heavily documentation oriented approaches like traditional methods, it is more important to focus on the working software that clearly and ”honestly” points out the facts about the progress and the success (or flaws) of the project and the team.

According to Cockburn (2001), ”*Documents can be very useful, but they should be used along with the words ”just enough” and ”barely sufficient.*” [7, p. 179], while the running code and the working software solidly show what has been

made.

Customer Collaboration over Contract Negotiation

The meaning of collaboration is to work on the same task together with active communication and by making decisions jointly. Hence, unlike the traditional methods that separate the development team from the customer as the end-user, Agile Software Development aims to obtain a close relationship between the team and the customer with strong collaboration.

Cockburn (2001) states that there is no "us" and "them" in Agile Software Development, there is only "us", while he describes the concept as "*Collaboration deals with community, amicability, joint decision making, rapidity of communication, and connects to the interactions of individuals.*". [7, p. 179]

On the other hand, customers are expected to have sufficient knowledge of the subject and to have the interest in attending to software development phase.

According to Boehm (2002), "*Unless customer participants are committed, knowledgeable, collaborative, representative, and empowered, the developed products generally do not transition into use successfully, even though they may satisfy the customer. Agile methods work best when such customers operate in dedicated mode with the development team, and when their tacit knowledge is sufficient for the full span of the application.*" [6, p. 66] Thus, in such cases traditional methods with contracts are more suitable.

Responding to Change over Following a Plan

This value supports the idea that the complete requirements for a software product cannot be known before it is used, due to the constant change of requirements, the development of information systems and the new business forms. In other words, it defends that rather than following a plan rigorously, it is better to flexibly respond to changes

according to customer's needs and priorities with periodical breaks.

As Cockburn (2001) states, building a plan is useful, and each of the agile methodologies such as SCRUM and Adaptive Software Development, has specific planning activities, while also containing mechanisms for dealing with changing priorities. [7, p. 179]

1.2.3 Principles behind the Agile Manifesto

In 2002, along with the four main values Agile Alliance [5] has published *The Twelve Principles behind the Agile Manifesto* that further explicate what it is to be Agile. These principles are as follows:

1) Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

2) Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

3) Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

4) Business people and developers must work together daily throughout the project.

5) Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

6) The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

7) Working software is the primary measure of progress.

8) Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain

a constant pace indefinitely.

9) Continuous attention to technical excellence and good design enhances agility.

10) Simplicity–the art of maximizing the amount of work not done–is essential.

11) The best architectures, requirements, and designs emerge from self-organizing teams.

12) At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly. [5]

These principles help us comprehend the properties of different Agile Development methods better and easier in the following chapter.

1.3 Different Styles of Agile Software Development

There are several different approaches of Agile Software Development that focus on different aspects of the projects. In this chapter, four of the most common ones of these styles, namely *eXtreme Programming*, *SCRUM*, *Adaptive Software Development*, *Crystal*, are presented.

At the end of the chapter, a comparison is made between the different Agile Styles, by also including other Agile Methods and their key points, namely *Dynamic System Development Method (DSDM)*, *Feature Driven Development (FDD)* and *Pragmatic Programming (PP)*.

1.3.1 eXtreme Programming (XP)

Extreme programming was originally started to be formulated in 1996 by Kent Beck as he states that extreme programming was invented as a reaction to the problematic

issues of the traditional methods with long and heavy development cycles. [3].

One of another important developers of extreme programming idea, namely Ron Jeffries, describes the method as "*Extreme Programming is a discipline of software development with values of simplicity, communication, feedback and courage. We focus on the roles of customer, manager, and programmer and accord key rights and responsibilities to those in those roles.*". [13, p. 9]

In contrast to the traditional methods, XP is based on small releases that are produced periodically, while it places much importance on customer satisfaction in parallel with continuous feedback, and thus in XP adopting changes of specifications is significant. Therefore, this naturally implies that the testing to obtain satisfying working releases plays a very crucial role in XP.

Jeffries summarizes the practices of XP as *On-site Customer*, *Small Releases*, *Planning Game*, *Metaphor*, *Simple Design*,*Pair Programming*, *Collective Code Ownership*, *Continuous Integration*, *Coding Standard*, *Test-Driven Development*, *Refactoring* and *40-Hour Week*. [13, p. 196]

These practices help us understand the principles of XP more precisely.

On-site Customer

XP requires the customers to actively and collaboratively participate to the project at all times. In this way, the team gains the constant availability of the customer that can always quickly provide instructions and answers about the requirements to the development team.

In XP, user stories written on cards that specify the requirements of the customers are used as core of the planning stage and while working on them the communication with the customer plays an important role, since the team wants to drive the project as fast as possible and to build the best that properly fits to what is required. [13, p. 31]

Small Releases

In XP, a project is developed by iteratively putting small-but-tested and -working releases that are updated periodically. In this manner, these small releases lead to the early benefit and/or use to the customer, thus to gain early feedback from the customer. [13, p. 65]

Planning Game

In XP, the scope of the next release is quickly determined according to specifications and priorities. Unlike traditional methods planning does not only depend on one plan determined at the beginning. About the rule of iteratively planning, Jeffries states that "*Inside each release, an Extreme team plans just a few weeks at a time, with clear objectives and solid estimates.*". [13, p. 79]

Furthermore, XP planning works continuously and iteratively considering the scheduling on the small releases and the goals for the next releases, according to the customer. Thus, this rule leads the customer to steer the development team by choosing the ideal combination of stories within the time and the funds available. [13, p. 59]

Metaphor

Metaphor rule aims to define and guide the development with a simple common story to ensure each member of the development team is completely aware of how the entire product works.

Cockburn expresses that a good metaphor helps the associations created around it turn out to be more appropriate to the development team. [7, p. 196]

Simple Design

XP developers stress implementing the simplest possible solution always in all stages. [9, p. 33] Hence, XP avoids the complexity and extra code. Hence, in XP the project is designed in a way as simple as possible. Beck expresses this idea as "Developers are urged to keep design as simple as possible, say everything once and only once. [3]

Pair Programming

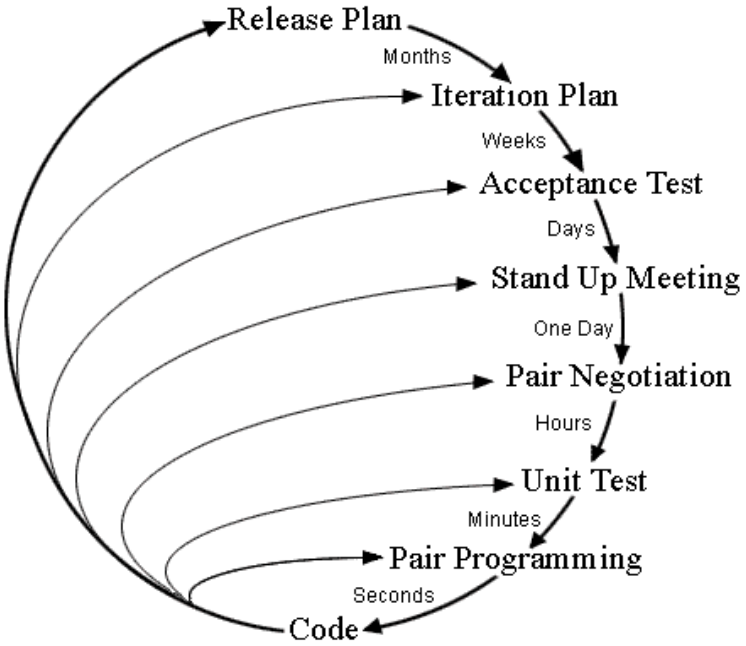

Figure 1.2: Planning and Feedback Loop in eXtreme Programming [21]

Pair programming is one of the most important rules of XP with two developers of the XP team working at the same computer.

Cockburn lists the advantages of pair programming as follows:

- Many mistakes get caught as they are being typed in rather than in Quality Assurance test or in the field (continuous code reviews)
- The end defect content is statistically lower (continuous code reviews)
- The designs are better and code length shorter
- The team solves problems faster
- The people learn significantly more, about the system and about software development
- The project ends up with multiple people understanding each piece of the system
- The people learn to work together and talk more often together, giving better information flow and team dynamics

[8]

Collective Code Ownership

In XP, the term collective ownership of a project means that each part of the code belongs to the whole development team. Thus, needed improvements on other programmers' code can be made by any member of the team while this also leads to a faster progress and cleaner code. [13, p. 145]

Coding Standard

In XP, a certain coding standard is used in order to have a common understanding and ability to work on the development. Jeffries supports this rule as *"Coding standard ensures that the code communicates as clearly as possible and supports our shared responsibility for quality everywhere."*. [13, p. 98]

Continuous Integration

"The longer we wait between integration and acceptance tests, the worse things get. Wait twice as long and we'll have four or more times the hassle. The reason is that one bug written just yesterday is pretty easy to find, while ten or a hundred written weeks ago can become almost impossible." [13, p. 97], Jeffries emphasizes on the importance of continuous integration.

Integration of the code is extremely difficult if it is done once at the of the entire development, due to many lines of code and the difficulty of identifying bugs. Thus, in XP each completed task is integrated to the system right away, then the application is built and tested daily several number of times. In this manner, the system always stays as completely integrated.

Test-Driven Development

In XP, testing is continuous and applied to each of the new releases frequently to ensure it works totally well with the whole system. XP testing consists of the unit tests applied by the development team and the acceptance tests made by the customer.

The unit tests are kept in an automated test suite by the programmers; and whenever they change a section of code, the test suite is run to observe immediately whether it caused a problem on what had been working, while the customer evaluates the new parts and give feedback right away. [7, p. 61]

Refactoring

Refactoring is the process of improving the structure of the code without changing its function. [13, p. 95] Thus, refactoring aims to keep the design of the code as simple and understandable as possible, to avoid duplication, and to add flexibility to the code.

40-Hour Week

This rule is opposed to working overtime defending that 40-Hour week should be the maximum time for the programmers to spend on the development, due to an ideal efficiency and concentration. Moreover, it expresses that the team should not work overtime two weeks in a row.

Jeffries supports the idea that heavy overtime is bad and suggests the rule as *"Do not work more than one consecutive week of overtime."*. [13, p. 101]

1.3.2 SCRUM

Ken Schwaber (1996), the pioneer of SCRUM, states that the development is an unpredictable process, whereas SCRUM produces breakthrough productivity, enabling building the best systems possible in complex, unpredictable environments.[17] Schwaber defines SCRUM as: *"Scrum is a method that aims to help teams to focus on their objectives. It tries to minimize the amount of work people have to spend tackling with less important concerns. Scrum is a response to keep things simple in the highly complicated and intellectually challenging software business environment."*[18, p. 35]

SCRUM consists of short, intensive, daily meetings of the whole project team aiming to deliver as much quality soft-

ware as possible within a series of short timeboxes called ”sprints”, which last about a month[19] .

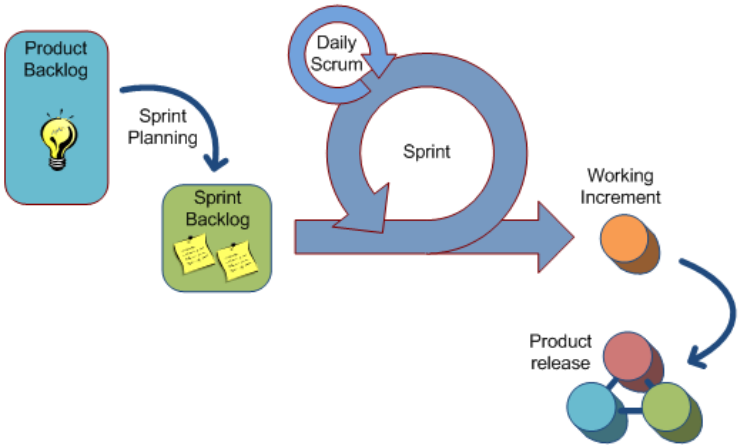

Figure 1.3: SCRUM Process Stages [20]

In SCRUM, each sprint has one certain goal determined by the development team using the sprint backlogs, which are obtained according to the product backlogs with requirements and priorities that the customer defined. Hence, sprints avoid the changes in order to keep the focus on the goal within the same task. The progress of the tasks are daily tracked visually by the development team and at end of each sprint the result is discussed and evaluated together.

In 1996 in his article *Controlled Chaos: Living on the Edge*, Schwaber lists the key principles of SCRUM as follows:

- Small working teams that maximize communication, minimize overhead, and maximize sharing of tacit, informal knowledge

- Adaptability to technical or marketplace (user/customer) changes to ensure the best possible product is produced

- Frequent ”builds”, or construction of executable, that can be inspected, adjusted, tested, documented, and built on

- Partitioning of work and team assignments into clean, low

coupling partitions, or packets

- Constant testing and documentation of a product - as it is built

- Ability to declare a product "done" whenever required (because the competition just shipped,because the company needs the cash, because the user/customer needs the functions, because that was when it was promised...) [17]

1.3.3 Adaptive Software Development (ASD)

In 1992, Jim Highsmith's effort of working on a short-interval, iterative, rapid application development process evolved into Adaptive Software Development (ASD), then in the mid- 1990s his interest in complex adaptive systems began to add a conceptual background to the team aspects of the practices. [12, p. 173]

ASD is an agile method that is based on the continuous change, and is opposed to stable planning, such as Waterfall approach's planning stage. ASD's change-oriented life cycle consists of three main stages as *Speculate*, *Collaborate* and *Learn*.

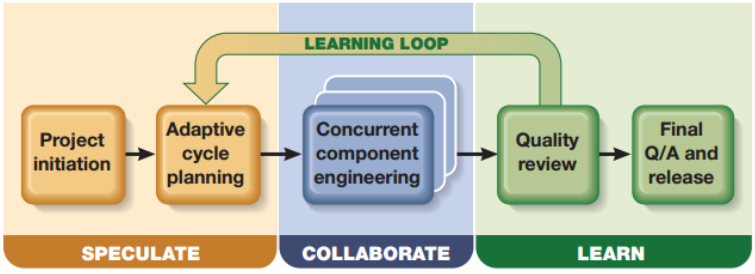

Figure 1.4: Adaptive Software Development Change-Oriented Life Cycle [11]

Speculate

Speculate stage of ASD's life cycle is used for initiation and cycle planning, as Highsmith explains it with the following seven steps:

1. *Conduct the project initiation phase.*
2.*Determine the project time-box.*
3. *Determine the optimal number of cycles and the time-box for each.*
4.*Write an objective statement for each cycle.*
5. *Assign primary components to cycles.*
6. *Assign technology and support components to cycles.*
7. *Develop a project task list.* [11, p. 26]

Collaborate

Working software is delivered by concurrent component engineering in ASD where interaction of people and management of interdependencies are crucial for the development as in XP's pair programming and collective code ownership methods. [11, p. 27]

Learn

Learning is an iterative step in ASD applied at the end of each development cycle, leading to a loop to adaptive planning for the next cycle and to quality assurance. Highsmith expresses that result quality from both the customer's perspective and a technical perspective are learned in this stage, as well as the functioning of the delivery team and the practices they are utilizing with project's status. [11, p. 27]

As mentioned in The Agile Manifesto's Working Software over Comprehensive Documentation value, ASD life cycle focuses on results which are identified as application features (customer functionality) that is to be developed during an iteration whereas the documentation is secondary. [12, p. 175]

1.3.4 Crystal

Crystal is a family of human-oriented light-weight methods with efficiency and agility purposes, developed by Alistair Cockburn in early 1990s. Highsmith states that Crystal focuses on collaboration and cooperation using project size,

criticality, and objectives to craft appropriately configured practices for each member of the Crystal family of methodologies. [12, p. xvi]

The design principles of Crystal can be summarized as:

The team can reduce intermediate work products as it produces running code more frequently, as it uses richer communication channels between people.

Every project is slightly different and evolves over time, so the methodology, the set of conventions the team adopts, must be tuned and evolve. [12, p. 144]

The figure The Family of Crystal Methods shown according to the number of people in the development team to the criticality by comfort, discretionary money, essentially money and life aspects.

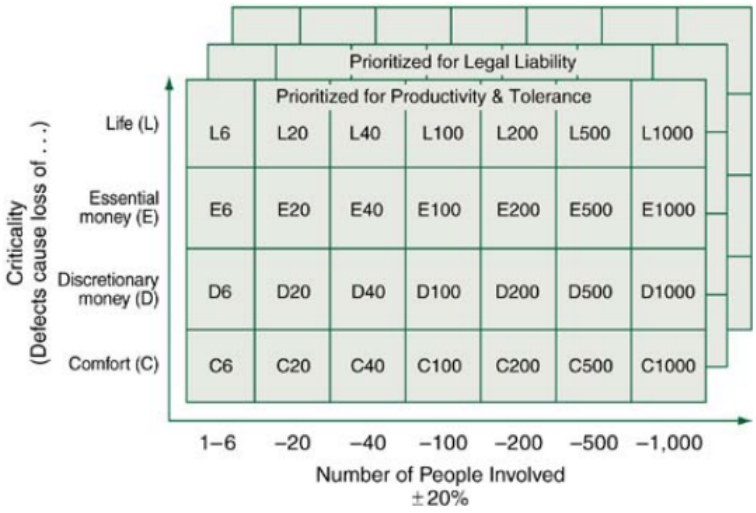

Figure 1.5: The Family of Crystal Methods [12, p. 146]

Crystal Clear is a member of Crystal family that is intended to be used by small project teams.

Tables 1.1 and 1.2 compare the different styles of Agile Software Development in terms of Key Points, Features, and Identified Shortcomings.

1.4 Agile Software Development vs. Waterfall Model

Using the knowledge covered and by reminding how the traditional *Waterfall Model* works, it is beneficial to compare the Agile Software Development to Waterfall approach in order to see how Agile Software Development tries to overcome shortcomings of traditional approaches. Table 1.3 shows the general differences between Agile Software Development and the traditional methods, at the end of this chapter.

Waterfall life cycle development is a sequential traditional model that has five separate main stages as *Requirements*, *Design*, *Implementation*, *Testing (Verification)* and *Maintenance* in order. In Waterfall approach, each step requires the previous one to be done; in other words the output of one step becomes the input for the next one. Furthermore, separate teams are responsible for these levels, due to being plan-driven and highly deadline-oriented.

In Waterfall approach, the linear progress of development basically works as: Initially the **requirements** and priorities are analyzed; then in the **design stage** the appropriate technology is determined according to these business requirements; **coding** the specified output of previous steps; **testing** the implementation; the final **evaluation** to ensure that the product works properly from top to bottom.

Flexibility for a Change

Since the Waterfall Approach is linearly stage dependent, it is not allowed or possible to work on previous stages that are already done. Thus, in Waterfall approach making changes is difficult, inefficient and costly, since it requires the system to be analyzed, designed and written from the beginning.

In contrast, as mentioned in *Responding to Change* value of *The Agile Manifesto*, Agile Software Development method

has the flexibility to adopt to change during each period of the development process.

Working Releases

In Waterfall approach, a working final software can be obtained, only after the last testing and evaluation phase on the entire product. Thus, in case any problematic issues occur, such as bugs, the whole software should be written from the beginning to fix.

In Agile Software Development, testing is made and the result is fixed after each step of the development process, then a second check is done on the related stage in order to obtain a running software constantly.

In the common sense, in Waterfall approach crucial problems may cause huge delays for the project that displeases the customer. Yet, in the Agile method there is an running software that exists for all the time, in order to appropriately develop the entire product in the targeted way.

Change of Customer's Needs

The Agile Software Development provides a flexible environment for the change of customer expectations during the development. Hence, the specifications of the project are possibly subject to change without difficulty, whereas the Waterfall approach requires the change of the entire project from the start due to being heavily documentation-oriented and passive customer communication.

1.5 Conclusion and Outlook

Rapidly changing technological and business environment played a role in resulting the traditional methods to become heavyweight and inefficient for most of the production based projects.

The Agile Manifesto is considered as a milestone for the development of new agile methodologies, since it provided a general base with its four values pointing out the most crucial facts of a software development process.

The process or role oriented traditional methods have difficulties, since individuals are not, in fact, replaceable; especially where the individuality is important as in software development. Furthermore, rather than focusing on strict and long-term plans, Agile Software Development focuses on responding to changes and on the working software with less documentation.

In conclusion to all, according to your author's opinion, Agile Software Development methods are more realistic, "sincere" and exciting due to the team work and continuous communication with clear minds, to develop projects .

Table 1.1: Comparison between Agile Software Development Styles [16, p. 89]

Method Name	Key Points	Special Features	Identified Shortcomings
XP	Customer driven development, small teams, daily builds	Refactoring - the ongoing redesign of the system to improve its performance and responsiveness too change	While individual practices are suitable for many situations, overall view and management practices are given less attention
SCRUM	Independent, small,self-organizing development teams, 30-day release cycles.	Enforce a paradigm shift from the defined and repeatable to the new product development view of Scrum	While Scrum details in specific how to manage the 30-day release cycle, the integration and acceptance tests are not detailed
ASD	Adaptive culture, collaboration, mission-driven component based iterative development	Organizations are seen as adaptive systems. Creating an emergent order out of a web of interconnected individuals	ASD is more about concepts and culture than the software practice
Crystal	Family of methods. Each has the same underlying core values and principles. Techniques, roles, tools and standards vary.	Method design principles. Ability to select the most suitable method based on project size and criticality.	Too early to estimate: Only two of four suggested methods exist.

Table 1.2: Comparison between other Agile Software Development Styles [16, p. 90]

Method Name	Key Points	Special Features	Identified Shortcomings
FDD	Five-step process, object-oriented component (i.e. feature) based development.	Method simplicity, design and implement the system by features, object modeling	FDD focuses only on design and implementation. Needs other supporting approaches.
DSDM	Application of controls to RAD, use of timeboxing and empowered DSDM teams.	First truly agile software development method, use of prototyping, several user roles : ”ambassador”, ”visionary”and ”advisor”	While the method is available,only consortium members have access to white papers dealing with the actual use of the method
PP	Emphasis on pragmatism, theory of programming is of less importance, high level of automation in all aspects of programming.	Concrete and empirically validated tips and hints	PP focuses on important individual practices. However, it is not a method through which a system can be developed.

Table 1.3: Comparison between Agile and Traditional Methods [2, p. 35]

	Agile Methods	**Heavy Methods**
Approach	Adaptive	Predictive
Success Measurement	Business Value	Conformation to plan
Project size	Small	Large
Management Style	Decentralized	Autocratic
Perspective to Change	Change Adaptability	Change Sustainability
Culture	Leadership-Collaboration	Command-Control
Documentation	Low	Heavy
Emphasis	People-Oriented	Process-Oriented
Cycles	Numerous	Limited
Domain	Unpredictable, Exploratory	Predictable
Upfront Planning	Minimal	Comprehensive
Return on Investment	Early in Project	End of Project
Team Size	Small/Creative	Large

Bibliography

[1] Environmental & social report. Technical report, Toyota Motor, 2003.

[2] M. A. Awad. A comparison between agile and traditional software development methodologies. Master's thesis, The University of Western Australia, 2005.

[3] K. Beck. Embracing change with extreme programming. *Computer*, 32(10):70–77, Oct. 1999.

[4] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas. Manifesto for agile software development, 2001.

[5] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas. The twelve principles of agile software, 2001.

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

[7] A. Cockburn. *Agile Software Development*. Addison-Wesley Professional, 2001.

[8] A. Cockburn and L. Williams. The costs and benefits of pair programming. In *In eXtreme Programming and Flexible Processes in Software Engineering XP2000*, pages 223–247. Addison-Wesley, 2000.

[9] M. L. David Cohen and P. Costa. Agile software development. Technical report, Data and Analysis Center for Software, 2003.

[10] S. L. Goldman, K. Preiss, and R. N. Nagel. *Agile Competitors and Virtual Organizations*. Van Nostrand Reinhold, 1995.

[11] J. Highsmith. Retiring lifecycle dinosaurs. *Software Testing & Quality Engineering (STQE)*, pages 22–28, 2000.

[12] J. Highsmith. *Agile Software Development Ecosystems*. Addison-Wesley Professional, 2002.

[13] R. E. Jeffries, A. Anderson, and C. Hendrickson. *Extreme Programming Installed*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[14] J. Liker and D. Meier. The toyota way fieldbook. 2005.

[15] J. K. Liker. *The Toyota way : 14 management principles from the world's greatest manufacturer / Jeffrey K. Liker*. McGraw-Hill, New York :, 2004.

[16] J. R. J. W. Pekka Abrahamsson, Outi Salo. Agile software development methods. VTT PUBLICATIONS 478, 2002.

[17] K. Schwaber. Controlled chaos: Living on the edge. *American Programmer 9*, 5:10–16, 1996.

[18] K. Schwaber. Against a sea of troubles: Scrum software development. *Cutter*, 13:34–39, 2000.

[19] J. Sutherland. Agile can scale: Inventing and reinventing scrum in five companies. *Cutter IT Journal*, 14:5–11, 2001.

[20] SYSART. Scrum diagram, http://www.sysart.fi.

[21] D. Wells. Extreme programming, http://www.xprogramming.com, 2000.

Typeset August 20, 2012