Agile Software Engineering

Ehsan Azizi Khadem[1], Emad Fereshteh Nezhad[2]

[1.] MSc of Computer Engineering, Department of Computer Engineering, Lorestan University, Iran
[2.] MSc of Computer Engineering, Communications Regulatory Authority of I.R of Iran
emad_fereshtehnejad@yahoo.com

Abstract: Rapid development of hardware in computer systems caused a deep hole between software and hardware usability and performance. Computer hardware has great growing in technology and tremendous increasing in speed and accuracy. New hardware segments are too smaller and electronic circuits that used in them are very integrated. Software must use capability of hardware in a computer system but software development is slower than hardware growing. When a team want to implement a software product, it consumes long time and human power. One of the reasons of this gap, is that we have not powerful methods to generate a software for several years. In 21th century, many software development methodologies presented for improving software producing. Although it was a great success and is saved time and software specialist effort, but unable to save time properly. Agile software engineering is used for solving this problem and improving time consuming of a software project production. It has several methods to implements rapid requirement capture, design, coding and testing a software project. in this paper, we explain about agile methods definition and characteristics and compare it with traditional methods in software engineering.

[Ehsan Azizi Khadem, Emad Fereshteh Nezhad. **Agile Software Engineering.** *World Rural Observ* 2018;10(1):31-36]. ISSN: 1944-6543 (Print); ISSN: 1944-6551 (Online). http://www.sciencepub.net/rural. 4. doi:10.7537/marswro100118.04.

Keywords: Agile, Software Engineering, Methodology

1. Introduction

A software system is the machine code, but what is machine code? It is a description in binary from that can read and understood by computer. A software system is a source code that is written by programmers that can be read and understood by a compiler. We can continue in this manner to ask similar questions about the design of a software system in terms of subsystems, classes, interaction diagrams, state chart diagrams and other artifacts. [1] They are part of the system. Requirements, testing, sales, production, installation and operations are part of the system too. A system is all the artifact that it takes to represent it in machine or human readable form to the machines, the workers, and the stakeholders. The machines are tools, compilers, or target computers. Workers include management, architects, developers, testers, marketers, administrators, and others. Stakeholders are the funding authorities, users, salespeople, project managers, line managers, production people, regulatory agencies, and so on. To support all of these artifacts, we use a software engineering methodology. A SE methodology direct all of process of software system developments in any steps. [2] It define a process that describes who is doing what when and how to reach a certain goal. In software engineering the goal is to build a software product or enhance an existing one. An effective process providing guidelines for the efficient development of quality software. It captures and presents the best practices that the current state of the art permits. In consequence, it reduces risk and increases predictability. The overall effect is to promote a common vision and culture. We need such a process to serve as a guide for all the participants, customers, developers, and executive managers. Any old process will not do. We need one that will be the best process the industry is capable of putting together at this point in its history. Finally, we need a process that will the widely available so that all the stakeholders can understand its roles in the development under consideration. A Software development process should be capable of evolving over many years. During this evolution it should limit reach at any given point in time to the realities that technologies, tools, people, and organizational patterns permit. Process must be built on technologies, programming languages, operating systems, computer systems, network capabilities, development environments, and so on that are usable at the time the process is to be used. For example, thirty years' age visual modeling was not really mainstream. It was too expensive. At that time, a process builder almost had to assume that hand drawn diagrams would be used. The assumption greatly limited the degree to which a process originator could build modeling into the process. Process and tools must develop in parallel. Tools are integral to process. To put it another way, a widely used process can support the investment that creates

the tools that support it. A process builder must limit the skill set needed to operate the process to the skills that current developers possess or target ones that developers can be quickly trained to use. In many areas it is now possible to embed techniques that once required expensive skill, such as checking model drawings for consistency, in computer-based tools. While software developers may not be as independently expert as symphony musicians. The process builder has to adapt the process to today's realities the facts of virtual organizations. Working at the distance through high speed lines. The mix of partial owners, salaried employees, contract workers, and outsourcing subcontractors and continuing shortage of software developers. Process engineers need to balance these four sets of circumstances. Traditional processes and methodologies like SSADM and RUP present frameworks for do all of these activities but they may use for long term project for at least six months. Today by increasing requests for mobile and web application we need a rapid manner to develop a software system and release it. Agile methods provide this for us. [3]

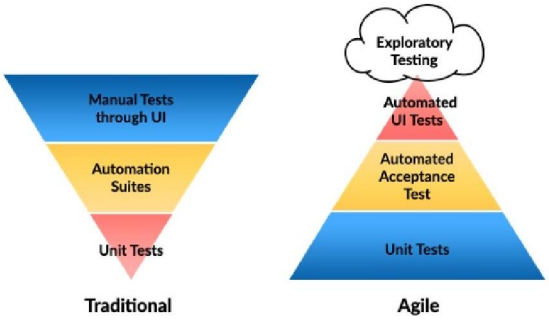

Figure 1: Traditional vs Agile Process

2. What is Agile Software Engineering?

Engineered and other systems are under pressure to adapt, from opportunities or competition, predators, changing environment, and physical or cyberattack. Ability to adapt well enough as conditions change, especially in presence of uncertainty, is valued. Systems (including developmental and life cycle management) that adapt well enough, in time, cost, and effectiveness, are sometimes called "agile". As environmental change or uncertainty increase, agility can mean survival. [4]

Agile systems and agile systems engineering are subjects of an INCOSE 2015-16 discovery project, described elsewhere. This paper introduces the underlying MBSE-based Agile Systems Engineering Life Cycle Pattern being used to capture, analyze, and communicate key aspects of systems being studied. More than an ontology, this model helps us understand necessary and sufficient conditions for agility, different approaches to it, and underlying relationships, performance couplings, and principles. [5]

This paper introduces the framework, while specific findings about methods and practicing enterprises studied will be reported separately. Iterative and incremental software development methods can be traced back to 1957. Evolutionary project management and adaptive software development emerged in the early 1970s. During the 1990s, a number of lightweight software development methods evolved in reaction to the prevailing heavyweight methods that critics described as heavily regulated, planned, and micro-managed. These included: from 1991, rapid application development; from 1994, the unified process and dynamic systems development method (DSDM); from 1995, Scrum; from 1996, Crystal Clear and extreme programming (XP); and from 1997, feature-driven development. Although these originated before the publication of the Manifesto for Agile Software Development, they are collectively referred to as agile software development methods. At the same time, similar changes were underway in manufacturing [13] and aerospace. [6]

In 2001, seventeen software developers met at the Snowbird resort in Utah to discuss these lightweight development methods, among others Jeff Sutherland, Ken Schwaber, and Alistair Cockburn. Together they published the Manifesto for Agile Software Development. [7]

In 2005, a group headed by Alistair Cockburn and Jim Highsmith wrote an addendum of project management principles, the Declaration of Interdependence, to guide software project management according to agile software development methods. [8]

In 2009, a movement by Robert C Martin wrote an extension of software development principles, the Software Craftsmanship Manifesto, to guide agile software development according to professional conduct and mastery. [9]

In 2011 the Agile Alliance created the Guide to Agile Practices (renamed the Agile Glossary in 2016) an evolving open-source compendium of the working definitions of agile practices, terms, and elements, along with interpretations and experience guidelines from the worldwide community of agile practitioners. [10]

Agile software development refers to a group of software development methodologies based on iterative development, where requirements and solutions evolve through collaboration between self-organizing cross-functional teams. Agile methods or Agile processes generally promote a disciplined project management process that encourages frequent inspection and adaptation, a leadership philosophy

that encourages teamwork, self-organization and accountability, a set of engineering best practices intended to allow for rapid delivery of high-quality software, and a business approach that aligns development with customer needs and company goals. Agile development refers to any development process that is aligned with the concepts of the Agile Manifesto. The Manifesto was developed by a group fourteen leading figures in the software industry, and reflects their experience of what approaches do and do not work for software development. Read more about the Agile Manifesto. [11]

3. Agile Methodology Properties

Projects that exhibit agile development success seem to share several key characteristics that are summarized below. For some methodologies these correspond exactly with individual practices, whereas for other methodologies there is a looser correspondence. [12]

Agile software development methods have two main units of delivery: releases and iterations. A release consists of several iterations, each of which is like a micro-project of its own. Features, defects, enhancement requests and other work items are organized, estimated and prioritized, then assigned to a release. Within a release, these work items are then assigned by priority to iterations. The result of each iteration is working, tested, accepted software and associated work items. [13]

Agile development projects thrive on the rhythm or heartbeat of fixed-length iterations. The continuous flow of new running, tested features at each iteration provides the feedback that enables the team to keep both the project and the system on track. Only from the features that emerge from fixed-length ("time-boxed") iterations can you get meaningful answers to questions like "How much work did we do last month compared to what we predicted we would?" and "How much work did we get done compared to the month before?" and our personal favorite, "How many features will we really get done before the deadline?"

The cruelty of several tight, fixed deadlines within an agile development release cycle focuses everyone's mind. Face to face with highly-visible outcomes from the last iteration (some positive, some negative), the team finds itself focused on refining the process for the next iteration. They are less tempted to "gold-plate" features, to be fuzzy about scope, or to let scope creep. Everyone can actually see and feel how every week, every day, and every hour counts. Everyone can help each other remain focused on the highest possible business value per unit of time. [14]

The operating mechanics of an agile development process are highly interdependent.

Traditional Framework	Agile Framework
More Rigid and Directions coming Top to Down	Team may conduct dozens of experiments to see which works best
More Commanding and Controlling style of Leadership	Communication flowing freely between all the team members
Spoon Feeding; Management tell everyone "What to Do"	Self-Organizing; work is distributed with consensus by the Team itself
Planning Centric and Plan Driven	Plan could be very fluid in Agile world as Agile projects are more fluid then waterfall projects
Document Oriented and Document Driven	Cross functional, Self-Contained and Agile is ultimately pragmatic
Resistant to Change	Agile Projects are welcoming to change; true even changes are introduced late in the project

Figure 2: Traditional vs Agile Properties

Each day, the agile development team is planning, working on, and completing tasks while the software is being designed, coded, tested and integrated for customer acceptance. Each iteration, the team is planning, testing, and delivering working software. Each release, the team is planning, testing, and deploying software into production. In order to coordinate and successfully deliver in such a highly adaptive and productive process, team communication and collaboration are critical throughout the entire agile development process.

As the iterations go by the team hits its stride, and the heartbeat of iteration deadlines is welcomed, not dreaded. Suddenly, once the team gets the hang of it, there is time for continuous process improvement, continuous learning and mentoring, and other best practices.

Delivering working, tested features are an agile development team's primary measure of progress. Working features serve as the basis for enabling and improving team collaboration, customer feedback, and overall project visibility. They provide the evidence that both the system and the project are on track. [3]

In early iterations of a new project, the team may not deliver many features. Within a few iterations, the team usually hits its stride. As the system emerges, the application design, architecture, and business priorities are all continuously evaluated. At every step along the way, the team continuously works to converge on the best business solution, using the latest input from customers, users, and other stakeholders. Iteration by iteration, everyone involved can see whether or not they will get what they want, and management can see whether they will get their money's worth. [4]

Consistently measuring success with actual software gives an agile development project a very different feeling than traditional projects. Programmers, customers, managers, and other stakeholders are focused, engaged, and confident. [6]

Agile development methods focus rigorously on delivering business value early and continuously, as measured by running, tested software. This requires that the team focuses on product features as the main unit of planning, tracking, and delivery. From week to week and from iteration to iteration, the team tracks how many running, tested features they are delivering. They may also require documents and other artifacts, but working features are paramount. This in turn requires that each "feature" is small enough to be delivered in a single iteration. Focusing on business value also requires that features be prioritized, and delivered in priority order. [7]

Different agile development methodologies use different terminology and techniques to describe features, but ultimately they concern the same thing: discrete units of product functionality.

It is a myth that agile methods forbid up-front planning. It is true that agile methods insist that up-front planning be held accountable for the resources it consumes. Agile planning is also based as much as possible on solid, historical data, not speculation. But most importantly, agile methods insist that planning continues throughout the project. The plan must continuously demonstrate its accuracy: nobody on an agile project will take it for granted that the plan is workable. [8]

At project launch, the development team does just enough planning to get going with the initial iteration and, if appropriate, to lay out a high-level release plan of features. And iterating is the key to continuous planning. Think of each iteration as a mini-project that receives "just-enough" of its own planning. At iteration start, the team selects a set of features to implement, and identifies and estimates each technical task for each feature. Task estimation is a critical agile skill. This same planning process repeats for each iteration.

It turns out that agile development projects typically involve more planning, and much better planning, than waterfall projects. One of the criticisms of "successful" waterfall projects is that they tend to deliver what was originally requested in the requirements document, not what the stakeholders discover they actually need as the project and system unfolds. Waterfall projects, because they can only "work the plan" in its original static state, get married in a shotgun wedding to every flaw in that plan. Agile projects are not bound by these initial flaws. Continuous planning, being based on solid, accurate, recent data, enables agile projects to allow priorities and exact scope to evolve, within reason, to accommodate the inescapable ways in which business needs continuously evolve. Continuous planning keeps the team and the system honed in on maximum business value by the deadline. [2]

In the agile community, waterfall projects are sometimes compared to "fire and forget" weapons, for which you painstakingly adjust a precise trajectory, press a fire button, and hope for the best. Agile projects are likened to cruise missiles, capable of continuous course correction as they fly, and therefore much likelier to hit the targeted feature-set and date accurately. [1]

Continuous planning is much more accurate if it occurs on at least two levels:

- At the release level, we identify and prioritize the features we must have, would like to have, and can do without by the deadline.
- At the iteration level, we pick and plan for the next batch of features to implement, in priority order. If features are too large to be estimated or delivered within a single iteration, we break them down further.

As features are prioritized and scheduled for an iteration, they are broken down into their discrete technical tasks.

This just-in-time approach to planning is easier and more accurate than large-scale up-front planning, because it aligns the level of information available with the level of detail necessary at the time. We do not make wild guesses about features far in the future. We don't waste time trying to plan at a level of detail that the data currently available to us does not support. We plan in little bites, instead of trying to swallow the entire cow at once. [12]

Many agile development teams use the practice of relative estimation for features to accelerate planning and remove unnecessary complexity. Instead of estimating features across a spectrum of unit lengths, they select a few (3-5) relative estimation categories, or buckets, and estimate all features in terms of these categories. Examples include: [4]

- 1-5 days
- 1, 2, or 3 story points
- 4, 8, 16, 40, or 80 hours

With relative estimation, estimating categories are approximate multiples of one another. For example, a 3-day feature should take 3 times as long as a 1-day feature, just as a 40-hour feature is approximately 5 times as time-consuming as an 8-hour feature. The concepts of relative estimation and/or predefined estimation buckets prevent the team from wasting time debating whether a particular feature is really 17.5 units or 19 units. While each individual estimate may not be as precise, the benefit of additional precision diminishes tremendously when aggregated across a large group of features. The significant time and effort saved by planning with this type of process often outweighs any costs of imprecise estimates. Just as with everything else in an agile

project, we get better at it as we go along. We refine our estimation successively. [8]

If a feature exceeds an agreed maximum estimate, then it should be broken down further into multiple features. The features generated as a result of this planning ultimately need to be able to be delivered within a single iteration. So if the team determines that features should not exceed 5 ideal days, then any feature that exceeds 5 days should be broken into smaller features. In this way we "normalize" the granularity of our features: the ratio of feature sizes is not enormous. [9]

As opposed to spending weeks or months detailing requirements before initiating development, agile development projects quickly prioritize and estimate features, and then refine details when necessary. Features for an iteration are described in more detail by the customers, testers, and developers working together. Additional features can be identified, but no feature is described in detail until it is prioritized for an iteration.

With continuous testing we deterministically measure progress and prevent defects. We crank out the running, tested features. We also reduce the risk of failure late in the project. What could be riskier than postponing all testing till the end of the project? Many waterfall projects have failed when they have discovered, in an endless late-project "test-and-fix" phase, that the architecture is fatally flawed, or the components of the system cannot be integrated, or the features are entirely unusable, or the defects cannot possibly be corrected in time. By practicing continuous testing in agile development, we more easily avoid both the risk that this will occur, and the constant dread of it. [5]

At both the unit level and acceptance feature level, we write the tests as the code itself is written beforehand. The most agile of agile development projects strive to automate as many tests as possible, relying on manual tests only when absolutely necessary. This speeds testing and delivers software that behaves predictably, which in turn gives us more continuous and more reliable feedback. There is an emerging wealth of new tools, techniques, and best practices for rigorous continuous testing; much of the innovation is originating in the Test-Driven Development (TDD) community. [9]

When is a feature done? When all of its unit tests and acceptance tests pass, and the customer accepts it. This is exactly what defines a running, tested feature. There is no better source of meaningful, highly-visible project metrics.

We continuously refine both the system and the project. By reflecting on what we have done using both hard metrics like running, tested features and more subjective measures, we can then adjust our estimates and plans accordingly. But we also use the same mechanism to successively refine and continuously improve the process itself.

Especially at the close of major milestones (iterations, releases, etc.), we may find problems with iteration planning, problems with the build process or integration process, problems with islands of knowledge among programmers, or any number of other problems. We look for points of leverage from which to shift those problems.

We adjust the factory's machines, and acquire or invent new ones, to keep doing it a little better each release. We keep finding ways to adapt the process to keep delivering a little more value per unit time to the customer, the team, and the organization. We keep maturing and evolving, like any healthy organism.

Smaller agile development teams have been proven to be much more productive than larger teams, with the ideal ranging from five to ten people. If you have to scale a project up to more people, make every effort to keep individual teams as small as possible and coordinate efforts across the teams. Scrum-based organizations of up to 800 have successfully employed a "Scrum of Scrums" approach to project planning and coordination. [11]

With increments of production-ready software being delivered every iteration, teams must also be cross-functional in order to be successful. This means that an agile development team needs to include members with all of the skills necessary to successfully deliver software, including analysis, design, coding, testing, writing, user interface design, planning, and management. We need this because, again, each iteration is its own mini-project. [13]

Teams work together to determine how best to take advantage of one another's skills and mentor each other. Teams transition away from designated testers and coders and designers to integrated teams in which each member helps do whatever needs doing to get the iteration done. Individual team members derive less personal identity from being a competitive expert with a narrow focus, and increasingly derive identity and satisfaction from being part of an extraordinarily productive and efficient team. As the positive reinforcement accumulates from iteration to iteration, the team becomes more cohesive. Ambient levels of trust, camaraderie, empathy, collaboration, and job satisfaction increase. Software development becomes fun again. These outcomes are not guaranteed, but they are much likelier in well-managed agile development projects than elsewhere. [12]

4. Advantages of the Agile Methodology:

a. The Agile methodology allows for changes to be made after the initial planning. Re-writes to the

program, as the client decides to make changes, are expected.

b. Because the Agile methodology allows you to make changes, it's easier to add features that will keep you up to date with the latest developments in your industry.

c. At the end of each sprint, project priorities are evaluated. This allows clients to add their feedback so that they ultimately get the product they desire.

d. The testing at the end of each sprint ensures that the bugs are caught and taken care of in the development cycle. They won't be found at the end.

e. Because the products are tested so thoroughly with Agile, the product could be launched at the end of any cycle. As a result, it's more likely to reach its launch date. [11]

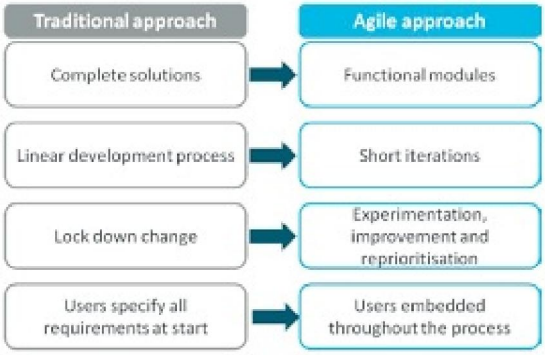

Figure 3: Advantage of Agile against Traditional Methodologies

5. Disadvantages of the Agile Methodology:

a. With a less successful project manager, the project can become a series of code sprints. If this happens, the project is likely to come in late and over budget.

b. As the initial project doesn't have a definitive plan, the final product can be grossly different than what was initially intended. [9]

6. Conclusion:

Software engineering methodologies is an important part of software system development. We must use them for reduce cost and time and increasing quality. A methodology determines workers, tasks, required diagrams and activities, amount of progress in project and so on. By using a methodology, we can determine time of any activity and test in a software development process. Traditional methodologies designed for generating a software in six months at least but today we must generate a software in short time. Agile software engineering methods can resolve this problem and allows for changes to be made after the initial planning. it's easier to add features that will keep you up to date with the latest developments in your industry but as the initial project doesn't have a definitive plan, the final product can be grossly different than what was initially intended.

References:

1. Javidi, M.M.; Kuchaki Rafsanjani, M. and Aliahmadipour, L. (2017). A Taxonomy of Game Theory Approaches for Intrusion Detection in MANETs. *Researcher*. Volume 9 - Issue 2. 88-96.
2. Hashim Ibrahim, B.; Adamu Yusuf, A. (2017). Optimizing Databases for High Performance and Efficiency. Report and Opinion. Volume 8 - Issue 9. 1-5.
3. Rothman, Johanna Rothman. "When You Have No Product Owner At All". www.jrothman.com. Retrieved 2014-06.
4. Fox, Alyssa. "Working on Multiple Agile Teams". techwhirl.com/. Retrieved 2014-06-14.
5. May, Robert. "Effective Sprint Planning". www.agileexecutives.org. Archived from the original on 28 June 2014. Retrieved 2014-06-14.
6. Berczuk, Steve. "Mission Possible: ScrumMaster and Technical Contributor". www.agileconnection.com. Retrieved 2014-06-14.
7. Namta, Rajneesh. "Thoughts on Test Automation in Agile". www.infoq.com. Retrieved 2014-06-14.
8. Band, Zvi. "Technical Debt + Red October". Retrieved 8 June 2014.
9. Shore, James. "The Art of Agile Development: Refactoring". www.jamesshore.com. Retrieved 2014-06-14.
10. Moran, Alan (2015). Managing Agile: Strategy, Implementation, Organisation and People. Springer. ISBN 978-3-319-16262-1.
11. Richet, Jean-Loup (2013). Agile Innovation. Cases and Applied Research, n°31. ESSEC-ISIS. ISBN 978-2-36456-091-8.
12. Newton Lee (2014). "Getting on the Billboard Charts: Music Production as Agile Software Development," Digital Da Vinci: Computers in Music. Springer Science+Business Media. ISBN 978-1-4939-0535-5.
13. Ebbage, Michael. "Setchu – Agile at Scale". Retrieved 30 September 2015.
14. Leybourn, Evan (2013). Directing the Agile Organisation: A Lean Approach to Business Management. IT Governance Publishing. ISBN 978-1-849-28491-2.
15. Sutherland, Jeff; Brown, Alex. "Scrum At Scale: Part 1". Retrieved 14 September 2015.
16. Beedle, Mike. "Enterprise Scrum". Retrieved 25 September 2015.