Volume 3, Issue 6, June 2013 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

An Empirical Study of Transition from Tayloristic Software Development Methods to Agile Environment

Manali Gupta*
Information Technology, ITS College, Noida, India

Neha Agarwal
Computer Science, Amity University, Noida, India

Abstract*— Software has been part of modern society for more than 50 years. Software development started off as a messy activity often mentioned as "code and fix". The software was written without much of a plan, and the design of the system was determined from many short term decisions. This worked well for small systems but as systems grew it became more difficult to add new features and bugs were harder to fix. This style of development was used for many years until an alternative was introduced: Methodology. Methodologies impose a disciplined process upon software development with the aim of making software development more predictable and more efficient.

Keywords*— Agile Testing, Extreme programming, Scrum, Tayloristic methods

I. INTRODUCTION

Software engineering is a knowledge-intensive process encompassing requirements gathering, design, development, testing, deployment, maintenance, and project coordination and management activities. It is highly unlikely that all members of a development team possess all the knowledge required for the aforementioned activities. More traditional approaches, like the Waterfall model and its variances, facilitate knowledge sharing primarily through documentation. They also promote usage of role based teams and detailed plans of entire software development lifecycle. The allocation of work specifies "not only what is to be done but how it is to be done and the exact time allowed for doing it" [1]. This shifts the focus from individuals and their creative abilities to the processes themselves. These traditional approaches are often referred to as "plan-driven" or "task-based". In contrary, agile methods emphasize and value individuals and interactions over processes. When comparing agile and traditional methods, the term "Tayloristic approaches" is used when discussing the traditional methodologies as the latter should not be referred as "plan-driven", because agile methods are also plan-driven. Lightweight methods that were implemented early include Scrum (1995), Crystal development techniques, Extreme Programming (1996), Adaptive Software Development, Feature Driven Development, and Dynamic Systems Development Method (DSDM) (1995). These are now typically referred to as agile methodologies, after the Agile Manifesto published in 2001.

Agile Manifesto [2] reads as follows:

"We are uncovering the better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- Individuals and interactions over processes and tools.
- Working software over comprehensive documentation.
- Customer collaboration over contract negotiation.
- Responding to change over following a plan.

II. AGILE MODELLING

Agile development is claimed to be a creative and responsive effort to address users' needs focused on the requirement to deliver relevant working business applications quicker and cheaper. [3]. The agile development approaches are typically concerned with maintaining user involvement through the application of design teams and special workshops. The delivered increments tend to be small and limited to short delivery periods to ensure rapid completion. Agile development is particularly useful in environments that change steadily and impose demands of early (partial) solutions. Agile approaches support the notion of concurrent development and delivery within an overall planned context. All the agile methodologies acknowledged that high quality software and more importantly customer satisfaction could only be achieved by bringing "lightness" to their processes.

A. Extreme Programming (XP)

Extreme programming (XP) has evolved from the problems caused by the long development cycles of traditional development models [4]. The XP process can be characterized by short development cycles, incremental planning, continuous feedback, reliance on communication, and evolutionary design. XP team members spend few minutes on programming, few minutes on project management, few minutes on design, few minutes on feedback, and few minutes

on team building many times each day. The term 'extreme' comes from taking these commonsense principles and practices to extreme levels.

The lifecycle of an XP project, shown in Fig. 1 [5], is divided into six phases: Exploration, Planning, Iterations to release, Production, Maintenance and Death. In the Exploration phase, the customer writes out the story cards they wish to be included in their program. This leads to Planning phase where a priority order is set to each user story and a schedule of the first release is developed. Next in the Iterations to Release phase, the development team first iteration is to create a system with the architecture of the whole system then continuously integrating and testing their code. Extra testing and checking of the performance of the system before the system can be released to the customer is done in the Production phase. Postponed ideas and suggestions found at this phase are documented for later implementation in the updated releases made at the Maintenance phase. Finally the Death Phase is near when the customer have no more stories to be implemented and all the necessary documentation of the system is written as no more changes to the architecture, design or code is made.

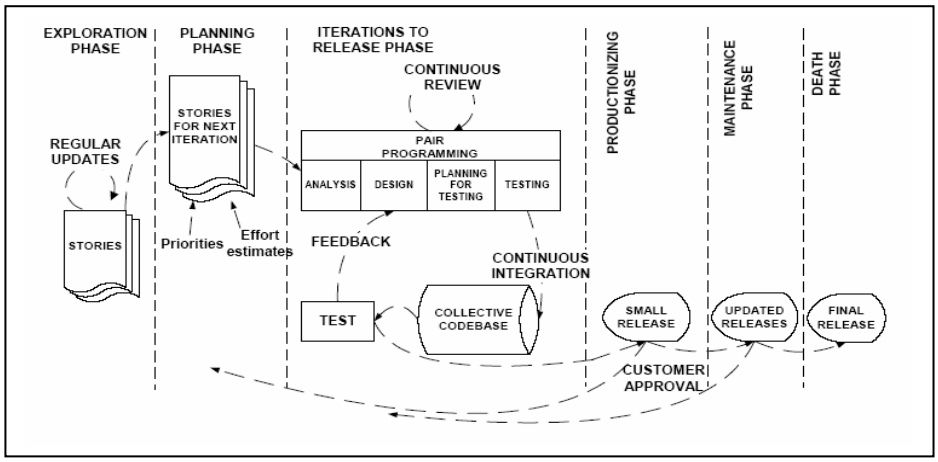

Fig. 1: Lifecycle of the XP process

B. Scrum

Scrum is an iterative, incremental process for developing any product or managing any work. Scrum concentrates on how the team members should function in order to produce the system flexibility in a constantly changing environment. At the end of every iteration it produces a potential set of functionality. Scrum does not require or provide any specific software development methods/practices to be used. Instead, it requires certain management practices and tools in different phases of Scrum to avoid the chaos by unpredictability and complexity [6]. The Scrum process may change the job description and customs of the Scrum project team considerably as shown in Fig. 2. For example, the project manager, i.e. the Scrum Master, does no longer need to organize the team but the team organizes itself and makes decisions on what to do. Rising and Janof [7] suggest that "Clearly, Scrum is not an approach for large, complex team structures, but we found that even small, isolated teams on a large project could make use of some elements of Scrum.

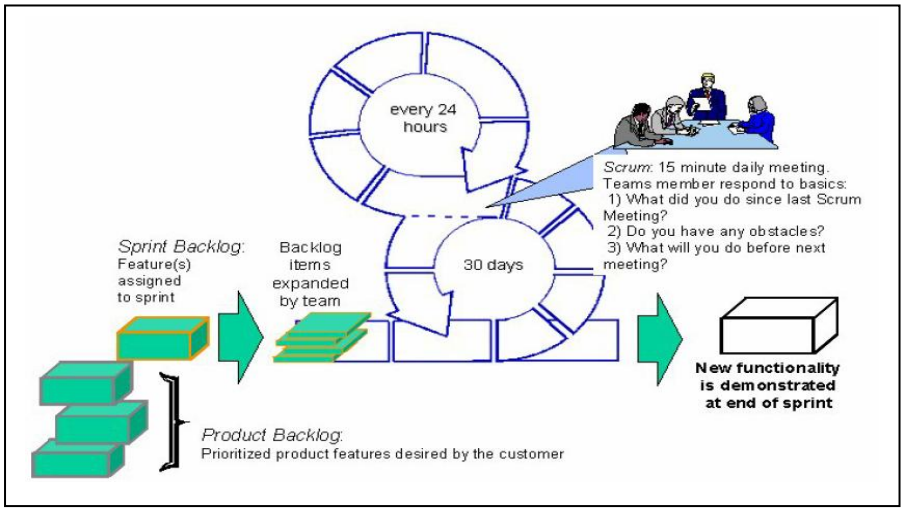

Fig. 2: Scrum Process

III. AGILE TESTING

Agile testing is a method of software testing that follows the principles of agile software development. In today's hasty world, stakeholders and customers want quick return on their investments. They don't want to wait for longer periods for a full featured product. As a result, nowadays new software testing paradigm is catching momentum, i.e., Scrum

approach. In agile scrum (sprint) process, projects are divided into small components to be developed and then to be tested in specific time-slice called as sprint (small cycles). Each feature should get developed and tested in a specified small time-slice. In the conventional development process which is based on phases, each phase goes through thorough a lengthy validation before triggering the next stage while Agile testing does not emphasize rigidly defined testing procedures, but rather focuses on testing iteratively against newly developed code until quality is achieved from an end customer's perspective. In other words, the emphasis is shifted from "Testers as Quality Police" to something more like "entire project team working toward demonstrable quality." [8] Agile teams test continuously. It's the only way to be sure that the features implemented during a given iteration or sprints are actually done. Continuous testing is the only way to ensure continuous progress. The time is measured between when a programmer writes a line of code and when someone or something executes that code and provides information about how it behaves. That's a feedback loop. If the software isn't tested until the very end of a long release, the feedback loops will be extended and can be measured in months. . Shorter feedback loops increase agility.

Agile projects favor automated tests because of the rapid feedback they provide. Manual regression tests take longer to execute and, because a human must be available, may not begin immediately. Feedback time increases to days or weeks. However, agile teams typically find that the fast feedback afforded by automated regression is a key to detecting problems quickly, thus reducing risk and rework. Therefore, agile testing refers to:-

- Individuals and interactions over processes and tools.
- Working software over comprehensive documentation.
- Customer collaboration over contract negotiation.
- Responding to change over following a plan.

Extreme Programming (XP) introduced the concept of pair programming. In Agile Testing, one can use pair testing together with the developer or another tester. One can also help developers design tests or create automated tests One of the first things an Agile Tester does is tuning the acceptance criteria and writing straight-forward Test Cases, which can be used by developers to drive their development. We call it Test Driven Development and this concept was already known from XP. The concept of TDD is very simple. First write a test, which will fail, since there is no feature yet and then write the part of the feature that is going to make the test pass. TDD can be used in the lowest layer of testing, which is Unit Testing but can be spread also to an Acceptance Testing layer. And again the automated test will originally fail and pass when the feature is developed. This approach is called Acceptance Test Driven Development. Acceptance Tests should be implemented as part of the Continuous Integration platform.

IV. TAYLORISTIC METHODOLOGIES

Tayloristic methodologies are considered to be the traditional way of developing software. These methodologies are based on a sequential series of steps, such as requirements definition, solution building, testing and deployment. Heavyweight methodologies require defining and documenting a stable set of requirements at the beginning of a project. Traditional methodologies are plan driven in which work begins with the elicitation and documentation of a complete set of requirements, followed by architectural and high level design development and inspection. Due to these heavy aspects, this methodology became to be known as heavyweight. Heavyweight methodologies have a tendency to first plan out a large part of the software process in great detail for a long span of time. This approach follows an engineering discipline where the development is predictive and repeatable. A lot of emphasis is put on the drawings focusing on the need of the system and how to resolve those needs efficiently. The drawings are then handed over to another group who are responsible for building the system. It is predicted that the building process will follow the drawings. The drawings specify how they need to build the system; it acts as the foundation to the construction process. The process would consist of certain tasks that must be performed by the managers, designers, coders, testers etc. For each of these tasks there is a well defined procedure. In the heavyweight environment, developers have the challenge to bring a seemingly infinite backlog of software projects, while keeping side by side of the latest advances. Survey after survey continues to prove that most software projects fail against some measure of success. Software are delivered late, over budget and do not meet the quality requirement. Furthermore, it is also difficult to research the causes of these failures.
However, typically, projects fail for one or more of the following reasons:-

- Requirements that are not clearly communicated
- Requirements that do not solve the business problem
- Requirements that change prior to the completion of the project
- Software (code) that has not been tested
- Software that has not been tested as the user will use it
- Software developed such that it is difficult to modify
- Software that is used for functions for which it was not intended
- Projects not staffed with the resources required in the project plan
- Schedule and scope commitments are made prior to fully understanding the requirements

V. AGILE METHODS VERSUS TAYLORISTIC METHODS

Agile methods are easier to use than traditional (heavyweight) methods because they include fewer instructions when analyzing, designing, or implementing the software requirements.

Heavyweight methodologies for software estimation and project planning work well if the requirements are clearly identified and they don't change. However, in most projects, the requirements do change during their lifetime, and therefore, developers need methodologies that can adapt well to the changing requirements. Agile methods permit a fast response to requirement changes since changes are considered as the rule, not the exception. In agile development, developers only need short project cycles. An executable system is not built near to the end of a project. Instead, it is built very early and delivered to the customer to be validated. Agile development is cooperative since customers and developers working constantly together with close communication. The customer is assumed to be present at the development site and is involved in the development process. Agile methodologies consider people – customers, developers, stakeholders, and end users – as the most important factor of software methodologies. Agile development methodologies truly give power to the developers. The developers make all the technical decisions, they make estimations for work to be done, they sign up for tasks for iteration, and they choose how much process to follow in a project. So, it is people-oriented rather than process-oriented [9]. Agile methods involve customer feedback on a regular and frequent basis. The customer of the software works closely with the development team, providing frequent feedback on their efforts. As well, constant collaboration between agile team members is essential. Agile methods develop software as an empirical (or nonlinear) process.

VI. MAIN ISSUES AFFECTING DECISION AND SELECTION OF METHODOLOGY

The agile and heavyweight methodologies both have their strengths and weaknesses. People usually follow either one of these methodologies or follow their own customized methodology. There are major factors affecting methodology decision and selecting which is suitable for various conditions. These factors can be categorized into project size, people and risk.

A. Project size

One of the limitations of agile methods is project size. The key elements of project size are project budget, duration and project team organization. The larger the team or more budget you need, the bigger the project is. Thus to compile more requirements, it requires more people and more coordination.

One of the founders of agile alliance claimed that for a given problem size, "fewer people are needed if a lighter methodology is used, and more people are needed if a heavier methodology is used" which is shown in Figure 3.

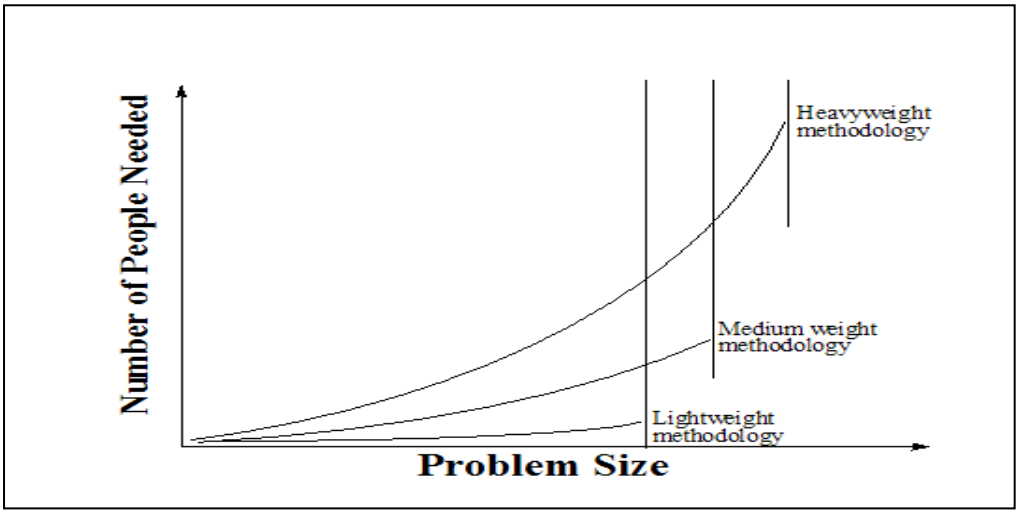

Fig. 3: Problem Size and Methodology Affecting Staff

B. People

Half of the agile manifesto values deal with human factors , "Individuals and interactions…" and "Customer collaboration." [10]. Having skill and experienced people in a team is a key factor for agile methodologies.

C. Risk factor

The most important risk factors in the development of a software process are project criticality and responding to change. Agile methods are used in applications that can be built quickly and do not require extensive quality assurance. Critical, reliable, and safe systems are more suited to a heavyweight methodology. If a project is critical, all requirements must be well defined before the development of the software. Poor definition would result in more damage from undetected defects. Responding to change can be resolved using an agile method. Practices defined in agile methods allow for better handling the changes, such as constant feedback from customer and short iterative development. With regards to software costs, adopting an agile process in comparison to a heavyweight process for a small-scale project will result in a decrease in cost. However, when it came to medium and large-scale projects the costs started to increase for adopting an agile process rather than a heavyweight process as shown in Figure 4.

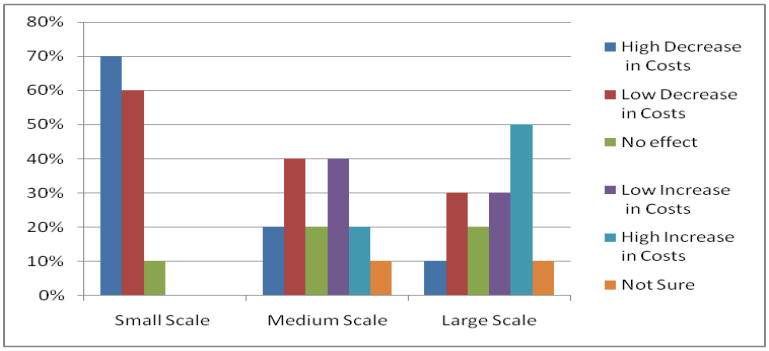

Fig. 4: Effect of Agile Methods on Cost

When the issue of software quality was presented the quality improvements follow heavyweight methodologies represented in Figure 5.

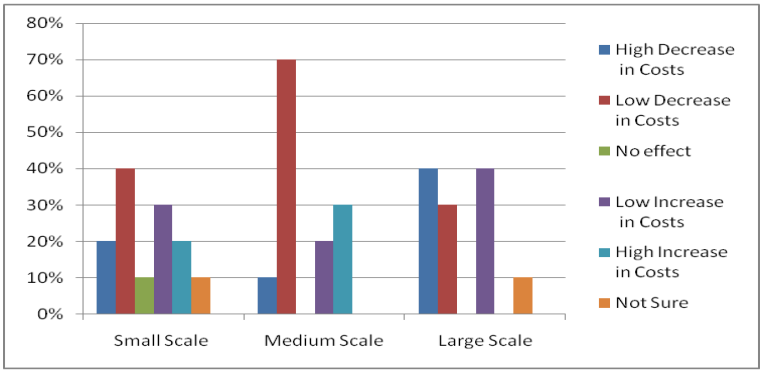

Fig. 5: Effect of Agile Methods on Quality

Figure 6 shows that lack of skilled people who can follow agile methodologies, was the major factor in both small and medium scaled projects. The major hurdle in using agile methods for large scale projects is project size and complexity. As the project size increases the number of people rises, thus increasing communication. Agile methodologies rely heavily on communication, so large teams make it difficult to use agile methods. There is a clear inverse relationship between agile techniques and project complexity.

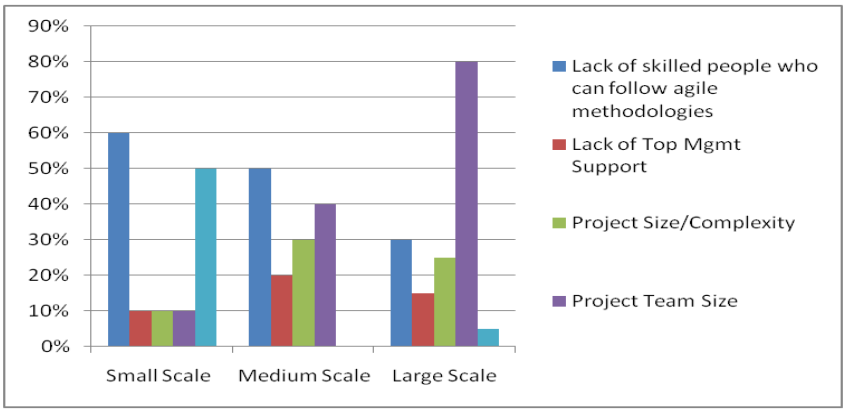

Fig. 6: Common Problem in agile methods

VII. CONCLUSION

Agile methodology is an alternative to plan-driven methodology (e.g.: UML or Waterfall Model) in the development of quality software. It is also very suitable for small to middle size projects because developers do not usually concentrate on models as the main component of the final software. In each software engineering phases, agile methods have unique approach that focus on feedback and change. Therefore, this method is believed can leverage the productivity of programmers. Just as mobile phones have reduced the need for telephone landlines agile methods are reducing the need for heavyweight methodologies. The future of agile methodologies seems very dominant. In general, there are some aspects of software development project can benefit from an agile approach and others can benefit from a more predictive traditional approach. When it comes to methodologies, each project is different. One thing is clear: that there is no "one-size-fits-all" solution.

Acknowledgment

Co-operation and co-ordination of various people at different levels go into a successful venture. It is with immense gratitude that we acknowledge the support and help of our faculty members and colleagues who have not only given us the valuable inputs but also were always ready with all their resource to help us out in the time of need.

References

1. F. Taylor, (1998) The Principles of Scientific Management, Dover Pubns, 1998.
2. Laraman, C., et al (2004) Agile and Iterative Development: A Manager's Guide. 2004, Boston MA: Addison-Wesley
3. K. Beck, Embracing change with Extreme Programming. IEEE Computer, (1999) Vol. 32, Issue 10 October 1999.
4. L. A. Williams, (2003) "The XP Programmer: The Few-Minutes Programmer", IEEE Software, pp.16-20, May/June 2003
5. K. Schwaber and M. Beedle, (2001) Agile Software Development with Scrum, Upper Saddle River, NJ, Prentice – Hall, 1st Edition, October 2001.
6. L. Rising and N. S. Janoff, (2000) The Scrum software development process for small teams, IEEE Software, Issue 17, pp. 26-32, 2000.
7. Helen Sharp, Hugh Robinson, Marian Petre: (2009)"The Role of Physical Arte facts in Agile Software Development: Two Complementary Perspectives", 2009.
8. J. Highsmith, et al (2002) Agile Software Development Ecosystem. Addison Wesley, 2002
9. A. Cockburn, J. Highsmith, (2001) "Agile Software Development: The People Factor", IEEE Computer, Vol. 34, No.11, 2001
10. Agile Manifesto. http://agilemanifesto.org

