CHAPTER 1

AGILE SOFTWARE DEVELOPMENT - AN INTRODUCTION

1.1 INTRODUCTION

Agile software development is a collection of software development methods by which requirements and their solutions would evolve through constant collaboration between cross-functional, self-organizing teams. The Agile development promotes adaptive planning, rapid development, short timeframe delivery, continuous improvement and response to change in a very rapid and flexible way (Moniruzzaman A., D.S.A. Hossain, 2013). To be competitive in today’s fast moving market place, organizations need to have determination to drive innovations in every part of the business. As a result more and more organizations are embracing agile development method as a viable development methodology that delivers customer values faster in both information technology and cross-key business units.

Agile is a development mindset focused on delivering customer value in relatively shorter timeframes. The work is continually shared with customers thus mitigating the risk of requirements misunderstanding and getting the validation done by the customer if the requirement is matching with their expectations. Agility provides highest business value is shortest time in an iterative, incremental manner.

The traditional software development methodologies fail to meet the current market scenarios. Nearly one thirds of the software projects are cancelled, three-fourths of the projects are considered failures by the persons who initiated them. Also, one in every two projects are overshooting the budget allocated to them. The main reason for the failures are feature shortfall of the expectations from customers. This is because most projects face the deadline rush and tend to cut the features that could not be achieved within budget and timelines. Also the projects need to be usable with more accuracy and speed of retrieval information. Even if the project delivers all the features as per the deadlines, but if they are not usable or if they do not meet the customer
expectations, then the project is a failure and has to be rewritten. The constant communication, incremental delivery, shorter timeframes, constant requirement verification and validation done in agile methodologies prove to be fruitful to deal with the scenario and avoid project failures.

1.2 MANIFESTO FOR AGILE SOFTWARE DEVELOPMENT

The Agile Manifesto was set down in February of 2001, at a meeting of seventeen independent-minded practitioners from several programming methodologies (Cockburn, Alistair, 2002). The participants came to consensus on four main values. These crucial guidelines and principles drive much of the activities that the teams do in agile software development. These guidelines are very difficult to understand and implement in the projects and are often misunderstood. The core idea can be understood as, to allow as much flexibility as possible in agile solutions and increase the chances of success. The manifesto for agile development is given as below.

**The Manifesto for Agile Software Development**

We are uncovering 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

That is, while there is value in the items on the right, we value the items on the left more.

Figure 1.2.1 Manifesto of Agile Software Development
The phrase "while there is value in the items on the right" should certainly be considered and cannot be ignored as the team delivers agile solutions.

**Individuals and Interactions over Processes and Tools**

IT professionals are required to deliver the software. The business users who orders the product to be built must describe it to the IT team that develops and delivers. Because of various factors prevailing, this development and delivery could become difficult from time to time. As the detailed processes and feature rich tools emerged, the focus slowly moved from the individuals to the processes and tools. But, this first point in the manifesto highlights and suggests that the focus on individuals and the effective communication between them should be of priority. The process and tools should be treated with the minimum importance for a situation. For a small company that develops a product with tight delivery timelines and resources, a simple, lightweight process with few required tools or no tools at all could be the appropriate solution. In this setup a large whiteboard and sticky notes could be the best solution for planning and progress tracking. For firms of mid and large size that have regulatory and auditing requirements, more process is required in terms of quality and audit. In these cases, formal tools and process may be necessary (Scott Ocamb, 2013).

The core objective of this point is the agile development team should use the light weight process and tools that are fairly required to do the needed tasks. It should be nothing less and nothing more and usage should be fair enough to get the required task done.

**Working Software over Comprehensive Documentation**

Projects that have concrete software requirements are likely to be the candidate in the failed projects. This is the model for the classic Waterfall approach. In this method, the delivery team talks to the customer and record the requirements in the early phases. The customer is then requested to go through the requirements and provide a sign off on them. Then the delivery team builds and tests the product defined in the requirements. There is a
huge time frame between the requirements sign off and the delivery of the project and a minimum interaction between the team and the customer once the development started. Also, it is nearly impossible for customers to document all the needed requirements in advance. Even if they do this, it's likely that the requirements would change between the time frame that the requirements are signed off and the product is delivered (Scott Ocamb, 2013).

The Agile approach of building potentially shippable chunks of the features in a product in relatively less time frames, is one of the most striking aspect and benefit an agile process can provide over the traditional processes. This approach constantly reveals the progress made so far and thus builds trust between the customer and delivery team. This benefit is because, useful software is constantly being delivered and the validations done more frequently. The customer also appreciates an evolving product and will notice things that never would have come to light if they stick on to the traditional ways of concrete requirements that would have been finalized in initial phases. The Agile way is to do the just required amount of activity, no more or less for a given task and the same is applicable for documentation. Some solutions need much more documentation and tractability matrices than others. Agile process can put up this whole range of requirements of the projects.

**Customer Collaboration over Contract Negotiation**

This is one among the most misinterpreted points of agile manifesto. This is not about saying that the projects don't require contracts and all collaboration should be informal. The meaning of this is that the iterative delivery method would have a better chance for delivering the needed product to the customer with the features he wants than to have a contract that needs to be adhered to and is tough to change. The contracts are required and also they should be flexible enough and have the appropriate change mechanism available to accommodate the agile process. The sales team should have to convince the
customer about the benefits of agile process and having a flexible approach in contract (Scott Ocamb, 2013).

**Responding to Change over Following a Plan**

The projects that have detailed project plans with thorough charts have been unsuccessful despite the concrete planning and execution. The plans turn out to be so complex and detailed that they turn to be tough to adapt to situations of incorporating changes. Agile process replaces project plans with release schedules and burn-down charts that can accommodate change with ease. The tracking of progress can be done easily and, in fact, progress is more transparent than in a typical Waterfall model. Agile processes are very transparent. The meaning of this is the progress on the project whether it is good or bad, occurs as a result of the process. There is little need for separately updating the tracking reports, burn down charts and maintaining them. To conclude, finally the results are the ones that matter in any project and the total purpose of the Agile Manifesto is to deliver better software and providing value to the customer. We need to focus on value and results that add to a firm’s competitive advantage in the current market. There are also examples of agile projects in which people forget the importance of the "items on the right “in the manifesto, which is not proper and would have adverse impacts. Using an Agile approach gives us permission to lessen the need for formality, but we should not ignore them (Scott Ocamb, 2013).

**1.3 THE PRINCIPLES OF AGILE SOFTWARE DEVELOPMENT**

The following 12 principles in addition to the four values when followed would lead to better outcome in the projects and better software. Understanding the principles is the key here and they seem to be quite contradictory, which makes them not so very easy to implement (Scott Ocamb, 2013), (Paul I. Pazderski, 2010).
**Principle 1:** Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

The highest priority is to satisfy the customer through timely and constant delivery of valued software. Team members can work together in a better way when they trust each other. Mutual trust can be accomplished when there is continuous and effective communication between the team. When the customer is happy by continuous delivery of software in short time cycles, trust and confidence on the team is built. In Scrum methodology, the customer for e.g., the product owner, decides what is really valuable and what needs prioritization. The product backlog is prioritized and the most valuable features are delivered first to gain customer confidence.

**Principle 2:** Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

In agile processes, the delivery of software is done frequently. Scrum is not an exception to this principle. Under Scrum, features are delivered in sprints of duration two to four weeks, with a preference of having the duration of two weeks.

**Principle 3:** Working software is the primary measure of progress.

Software delivered often and the features delivered should be valuable and, the delivered features must be working or done. Scrum requires the features to meet a team-defined definition of 'Done'. Ideally, this means that the identified feature is potentially shippable and customer can start using it.

**Principle 4:** Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

Agile team delivers software at the end of each iteration, a constant pace that is sustainable without overburdening anyone will surface. This allows the team to work continuously until enough value has been added to the product.
An important aspect of this is regular releases of a product. For instance, if a team can deliver a shippable product each quarter, it would help the conversations with the customer much easier. The team knows the fact that they ship every 12 weeks. When the customer requests for a feature, then the team can see the possibility if it is shippable in the current iteration or the future iterations coming next.

**Principle 5: Business people and developers must work together daily throughout the project.**

The whole team needs to be readily available to each other. Scrum uses the daily stand-up meeting as a critical communication mechanism. In the daily stand up meeting, the team reports

1) What has been accomplished since the last meeting?

2) What will be accomplished by the next meeting?

3) Are there are any impediments that are avoiding the team to complete the planned the features in the sprint?

This meeting uncovers the issues at an early stage, so that they can be addressed and corrective actions taken even before they become critical.

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

This principle was authored before Distributed agile teams were common. Today, with offshore teams and teams that are divided across the country and the globe, regular face-to-face communication is often not possible. Web conferences, On-line meetings and instant messaging tools are available that improve communication when teams are geographically dispersed. These teams can be called as Distributed Agile Teams (DAT). Meetings that include the whole team may be planned so that face-to-face communication is possible. This does add cost to the project, because portions of the team
need to travel to a central location for the meeting. This approach is helpful for important meetings like sprint and release planning. In the case of DAT, the web conferences with appropriate tools can be used to overcome the burden of travel costs, in the cases where the travel cost is considerable. Team rotation from offshore to onshore and vice versa also help to overcome the issue of distributed setup and allows team members to interact personally and get to know each other. It allows the offshore team to return home with firsthand experience and the feeling of oneness between the team that helps the remote team gain valuable insight of the project and people. If possible the arrangement should be done that individual teams are co-located. It is best to have face-to-face communication, and techniques like this should be used to get as close to this goal as possible. The teams need to communicate in order to be successful and the communication should be effective. Effective communication gives the scope for better understanding the team and building the mutual trust which is a key factor in the success of the projects.

**Principle 7: The best architectures, requirements, and designs emerge from self-organizing teams.**

The team knows the best way to get something done when equipped with the all relevant details like technicalities and functional scope. The term servant leader has arose in the agile community and displaced the typical command-and-control project manager. He or she asks probing questions to know the details on the areas that individual team members may not notice. It is critical for the servant leader to treat everyone with respect and dignity, even when pressures are high. A servant leader understands that conflict among team members is normal and embraces healthy debate. Self-organizing teams do not transpire automatically. They develop under the proper direction and guidance of a servant leader.
Principle 8: Build projects around motivated individuals. Give them the environment and support they need and trust them to get the job done.

This is an extension of self-organizing teams. The servant leader should have consideration to the aspirations and goals of the team members and should align these goals with project needs everywhere possible. It is a known fact that people performance is at their best when they are undertaking something they are passionate about. A good servant leader also shelters the team from outside distractions. In Scrum, a team commits to completing a set of features. Any factor that distracts the team from doing what they ought to do is a risk. It can be the product owner, business community or environmental issues that can distract the team from their regular work. The scrum master should provide the insulation to the team from these external factors and provides them with the environment and support needed for success. Trust is not automatic but is built over time -- and is easy to lose. The team members must trust each other and be comfortable with conflict. Here are some tips to building trust in a team:

- Be honest in admitting facts and faults. Do not intentionally hide any faults made.
- Do what you say. Stick on the timelines and perform the needed as committed.
- Follow up. Follow up on open issues that take some time. This allows people to see that you have not forgotten them.

Principle 9: At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

The retrospective is the perfect place for the team to reflect and improve. In Scrum a retrospective meeting would help in identifying areas for improvement that the team need to really improve. This can be from lessons learnt, be it a team communication and be it a coding practice or a better testing scenario. If teams spend time reflecting and do not improve, then the
retrospection is a waste of time. The Scrum Master should tactfully prompt the team about areas of improvement they agreed to. In some scrum teams, the points they agreed to up are pinned up on the wall where the stand-up meetings take place.

**Principle 10: Continuous attention to technical excellence and good design enhances agility.**

The Agile team need to have consideration to technical excellence and design as the product evolves. There should be a balance between "Building the right thing" and "Building the thing right." The team must also be wary of delivering fragile systems and try to avoid this. If the end product is not capable to sustain the addition of few changes then that is not a good sign.

Agile methodologies such as Extreme Programming and Scrum to an extent recommend test-driven development and automated builds as a way to prevent fragile solutions.

**Principle 11: Simplicity, the art of maximizing the amount of work not done, is essential.**

Agile is all about doing nothing less and nothing more than required at any given time. The user stories and its tasks should be small enough to get the job done and should not go beyond the needed.

**Principle 12: Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.**

This principle is not very much appreciated by the teams who are used to Waterfall projects. At first glance, it seems very unusual to welcome change late in the development process. The team should be successful at implementing the first two agile principles. If the team faces difficulties in implementing so then welcoming change is impossible. "Late in development" means late in the release of the complete product. Scrum delivers features in
short sprints of sizes two to four weeks and the changes are not welcome once the sprint is in-process. The change should then go to the next sprint possible. In Scrum, the change is directed by the product owner as he knows the competitive advantage of each feature in the backlog and the change thereof.

1.4 WORKING OF AGILE METHODOLOGIES

Iterative, Incremental and Evolutionary

Agile methods generally breaks the features into manageable tasks which require minimum planning. The agile methods contain iterations which are time boxed or time framed. These time boxes are generally of duration two to four weeks. Each iteration targets at delivering a potentially shippable set of features which can be validated and are usable by the customers. The incremental delivery model would help minimizing the overall risk of missing requirements, expectation mismatch and helps the project to quickly analyst and adapt to the changes that are required to be prioritized during the next iterations. The goal of an iteration is to have usable functionality with minimal bugs. A set of iterations can be called a release which would target a set of co-related features that can be released together. The product would finally be shipped once all its underlying planned releases are delivered without any hindrance, meaning all features of the product are released be it small or big, easy or complex (Beck, Kent ,1999).

The Agile projects would definitely have an active stakeholder from the business who orders the software. For instance in scrum the product owner is the person who would actively involve in scrim ceremonies and would have a visualization and roadmap of the overall idea of the project, its features and their rankings. The product owner would concentrate to groom up the product backlog during the time when the scrum team is busy handling their scrum tasks. However, during various ceremonies of scrum such as sprint planning,
retrospective meetings, daily stand up meetings etc., it is always advisable to have the team to seat together in a meeting room whenever it is possible. The face to face communication is of high importance and should not be sacrificed for mere discomfort of traveling from one floor to other floor. The teams tend to sit in multiple levels in a building and it’s often observed they tend to join the scrum daily stand up over a phone. If the possibilities are there to join a face to face meeting its better and would be more effective.

There are also many tools and techniques that are used to improve quality of the product and enhance projects agility. For e.g. pair programming in Extreme programming (XP) or UI programming (UIP) in Scrum methodology are few practices that improve the code quality.

Agile development is targeted at building complex systems which have volatile or dynamic requirements that constantly change over time and where concrete requirements, concrete estimates cannot be presented for driving the development and delivery of software. The treasurable industry experiences educated the teams over years on the success and failure criteria’s and this has helped agile software development to be driven towards adaptive, evolutionary and iterative nature of development.

**Some Popular Agile Methodologies for Software Development.**
The agile methods deals with different aspects in the software development life cycle. Some methods focus on the practices such as Pair Programming, Planning poker, Agile Modeling etc., while others deals with the execution and managing of the software projects for e.g. Scrum. Further, there are approaches providing full coverage over the development life cycle for e.g., DSDM, IBM RUP, while most of them are suitable from the requirements specification phase on for e.g. FDD, for example. Thus, there is a clear variance amongst the different agile methods in this regard.
1.4.1 SCRUM

Scrum is a lightweight process framework for agile development, and the most widely-used one. A process framework is a particular set of practices that must be followed in order for a process to be consistent with the framework. For example, the Scrum process framework requires the use of development cycles called Sprints, the XP framework requires pair programming etc. Lightweight means that the overhead of the process is kept as small as possible, to maximize the amount of productive time available for getting useful work done (Deepa Vijay, Gopinath Ganapathy 2014).

A Scrum process is distinguished from other agile processes by specific concepts and practices, divided into the three categories of Roles, Artifacts, and Time Boxes (CPrime, no date). These and other terms used in Scrum are defined below.
Scrum is most often used to manage complex software and product development, using iterative and incremental practices. Scrum significantly increases productivity and reduces time to benefits relative to classic waterfall processes. Scrum processes enable organizations to adjust smoothly to rapidly-changing requirements, and produce a product that meets evolving business goals.

**Benefits of Scrum**

*Benefits to Customer*

Customers find that the vendor is more responsive to development requests. High-value features are developed and delivered more quickly with short cycles, than with the longer cycles favored by classic “waterfall” processes (CPrime, no date).

*Benefits to Vendors*

Vendors reduce wastage by focusing development effort on high-value features, and reduce time-to-market relative to waterfall processes due to decreased overhead and increased efficiency. Improved customer satisfaction translates to better customer retention and more positive customer references.

*Benefits to Development Teams*

Team members enjoy development work, and like to see their work used and valued. Scrum benefits Team members by reducing non-productive work (e.g., writing specifications or other artifacts that no one uses), and giving them more time to do the work they enjoy. Team members also know their work is valued, because requirements are chosen to maximize value to customers.
Benefits to Product Managers

Product Managers, who typically fill the Product Owner role, are responsible for making customers happy by ensuring that development work is aligned with customer needs. Scrum makes this alignment easier by providing frequent opportunities to re-prioritize work, to ensure maximum delivery of value.

Benefits to Project Managers

Project Managers (and others) who fill the Scrum Master role find that planning and tracking are easier and more concrete, compared to waterfall processes. The focus on task-level tracking, the use of Burn down Charts to display daily progress, and the Daily Scrum meetings, all together give the Project Manager tremendous awareness about the state of the project at all times. This awareness is key to monitoring the project, and to catching and addressing issues quickly.

Benefits to PMOs and C-Level Executives

Scrum provides high visibility into the state of a development project, on a daily basis. External stakeholders, such as C-Level executives and personnel in the Project Management Office, can use this visibility to plan more affectively, and adjust their strategies based on more hard information and less speculation.

Scrum differs from traditional “waterfall” approaches to project management in many ways, but is based on sound project-management principles.
SCRUM ROLES

The three roles defined in Scrum are the Scrum Master, the Product Owner, and the Team (which consists of Team members). The people who fulfill these roles work together closely, on a daily basis, to ensure the smooth flow of information and the quick resolution of issues (CPrime, no date).

Scrum Master

The Scrum Master (sometimes written “Scrum Master,” although the official term has no space after “Scrum”) is the keeper of the process. He is responsible for making the process run smoothly, for removing obstacles that impact productivity, and for organizing and facilitating the critical meetings. The scrum masters responsibilities include

1) Removing the barriers between the development Team and the Product Owner so that the Product Owner directly drives development.

2) Teach the Product Owner how to maximize return on investment (ROI), and meet his/her objectives through Scrum.

3) Improve the lives of the development Team by facilitating creativity and empowerment.

4) Improve the productivity of the development Team in any way possible.

5) Improve the engineering practices and tools so that each increment of functionality is potentially shippable.

6) Keep information about the Team’s progress up to date and visible to all parties.

In practical terms, the Scrum Master needs to understand Scrum well enough to train and mentor the other roles, and educate and assist other stakeholders who are involved in the process. He should maintain a constant awareness of the status of the project (its progress to date) relative to the expected progress, investigate and facilitate resolution of any roadblocks that hold back
progress, and generally be flexible enough to identify and deal with any issues that arise, in any way that is required. He must protect the Team from disturbance from other people by acting as the interface between the two.

The Scrum Master does not assign tasks to Team members, as task assignment is a Team responsibility. His general approach towards the Team is to encourage and facilitate their decision-making and problem-solving capabilities, so that they can work with increasing efficiency and decreasing need for supervision. His goal is to have a team that is not only empowered to make important decisions, but does so well and routinely.

**Product Owner**

The Product Owner is the keeper of the requirements. He provides the “single source of truth” for the Team regarding requirements and their planned order of implementation.

In practice, the Product Owner is the interface between the business, the customers, and their product related needs on one side, and the Team on the other. He buffers the Team from feature and bug-fix requests that come from many sources, and is the single point of contact for all questions about product requirements. He works closely with the team to define the user-facing and technical requirements, to document the requirements as needed, and to determine the order of their implementation. He maintains the Product Backlog (which is the repository for all of this information), keeping it up to date and at the level of detail and quality the Team requires.

The Product Owner also sets the schedule for releasing completed work to customers, and makes the final call as to whether implementations have the features and quality required for release.
**Scrum Team**

Scrum Team refers to the Team plus the Scrum Master and Product Owner. The Team size should be kept in the range from five to nine people, if possible. This because, a larger number make communication difficult, while a smaller number leads to low productivity and fragility.

**Scrum Ceremonies**

The following are the ceremonies performed in the lifecycle of a normal Scrum:

- Sprint Planning Meeting
- Daily Scrum Meeting
- Sprint Review/Demonstration Meeting
- Sprint Retrospection Meeting

**Sprint Planning Meeting**

Time box: 4 hours (2 weeks sprint) / 8 hours (4 weeks sprint).

Attendees: Scrum Master, Product Owner and Scrum Team.

This meeting consists of two sections.

During the first section the Product owner describes and presents the Product Backlog items that need to be achieved and Sprint goal to the entire Scrum Team. The scrum team then collaborates about understanding the work and the ways to perform the Sprint.

In second section of this meeting scrum team plans in detail by creating work breakdown into manageable tasks and identifying the tasks that are essential to fulfill the Sprint Goal and deliver the projected Product Backlog items according to capacity of team.
Inputs to Sprint Planning Meeting: Team Capacity and Product Backlog

Output of Sprint Planning Meeting: Sprint Goal and Sprint Backlog

**Daily Scrum Meeting**

The Daily Scrum is the key inspect and adapt meeting during a Sprint. During the Sprint execution, the scrum Team meets every day, for Daily Scrum meeting and inspects the progress and ensures communication flow inside the Team. It is a short (15 minutes) meeting.

Time box: 15 Minutes.

Attendees: Scrum Master, Product Owner (Optional) and Scrum Team.

It is held at the same time at same place each day. During the meeting, each Team member explains

- What have I accomplished since the last meeting?
- What am I going to do before the next meeting?
- What obstacles are in my way?

The Daily Scrum improves communication, eliminates other meetings, identifies & removes impediments to development, highlights and promotes quick decision-making, and improves everyone’s level of project knowledge. The responsibility for conducting the Daily Scrum is with the Team and Scrum Master facilitates the same. The Scrum Master ensures the all the impediments are noted and he/she will get these resolved. He coaches the Team to keep the Daily Scrum short and making sure that people speak briefly.
Sprint Review/Demonstration Meeting

Time box: 1.5 hours -3 hours (2 - 4 weeks sprint).

Attendees: Scrum Master, Product Owner and Scrum Team, All the Stakeholder/Sponsors, Customers.

A Sprint Review/Demo meeting is held at the end of the Sprint to inspect the Increment. The Team demonstrates the Increment with focus on the Sprint Goal according to the Definition of Done. The Product Owner reviews and accepts the delivered Increment. During the Sprint Review, Product Owner, Team and stakeholders review what was done. This meeting should not have Slides, with the presentation of the results but should have working demonstration of the work planned in sprint planning.

After the demonstration the Product Owner and other relevant stakeholders tell their impressions and clarify their requirements (user stories) if a requirement was not implement right. The Product Owner identifies what has been done and what hasn’t been done (according to the Definition of Done). The Product Owner accepts the user stories that have been done. Results of this meeting can be new requirements in the Product Backlog, and a new prioritization of existing Product Backlog items.

Sprint Retrospection Meeting

Time box: 2 hours - 4 hours (2-4 weeks sprint).

Attendees: Scrum Master, Product Owner and Scrum Team.

In the Sprint Retrospective the Scrum Team revises their way of work in the past in order to make it more efficient and effective in the future. The Scrum Master encourages the Scrum Team to search for best practices and to identify improvement measures that it will implement in the next Sprint. After the Sprint Review and before the next Sprint Planning, the Team has a Sprint Retrospective. The Sprint Review is about the product, the Sprint
Retrospective is about the process – the way in which the Scrum team works. It is never omitted. In the Sprint Retrospective meeting, the Scrum Master encourages the Development Team to inspect, within the Scrum framework and practices, how the last Sprint went in regards to people, relationships, process and tools. The Team should identify and prioritize the major items that went well, and those items that, if done differently, could make things even better. By the end of the Sprint Retrospective, the Team should have identified actionable improvement measures that they will implement in the next Sprint.

**Scrum Artifacts**

There are following artifacts in Scrum.

- Product Backlog
- Sprint Backlog
- Product Increment

**Product Backlog**

The Product Backlog is an ordered list of everything that might be needed in the product and is the single source of requirements for any changes to be made to the product. The Product Backlog is dynamic; it constantly changes to identify what the product needs to be appropriate, competitive, and useful. It contains broad descriptions of all required features, functions, enhancements, wish-list items, etc. The Product Backlog defines the “What” that will be built.

The Product Backlog is the property of the Product Owner. He is responsible for its content, its availability and prioritization. Business value is set by the Product Owner. Development effort is set by the Team. All items in the
Product Backlog are prioritized and sorted by business value. The top priority items drive the next development activities. Higher priority items are clearer and have more detailed information than lower priority items. As Increments are being reviewed and releases of the product are being used, feedback is provided and the Product Backlog emerges into a larger, more exhaustive and more detailed list.

The Product Backlog items are initially established and calculated during Release Planning. Afterwards they are updated in Sprint Planning or backlog grooming.

The top priority items are selected for development during Sprint Planning developed in the Sprint and reviewed in the Sprint Review. Following figure shows the sample product backlog created within the tool.

![Sample Product Backlog](image)

**Figure 1.4.1.1 Product backlog sample (Source- Version one)**

**Sprint Backlog**

The Sprint Backlog is the set of Product Backlog items selected for the Sprint plus a plan for delivering the product increment and realizing the Sprint Goal. The Sprint Backlog is a forecast by the Team in sprint planning meeting and
is about what functionality will be in the next Increment and the work needed to deliver that functionality.

In the Sprint Backlog the Team plans the necessary tasks to implement the items selected from the Product Backlog in Sprint Planning. The Sprint Backlog is property of the Team. Estimations are set by the Development Team. The Team keeps the Sprint Backlog always up to date during the Sprint. As the work is done, the Development Team may find that more, less or different tasks are needed. The Sprint Burn Down graph shows the remaining effort across the time of a Sprint. Often a Task Board is used to see and change the state of the tasks of the current sprint, like “to do”, “in progress” and “done”.

Following are the figures showing a sample task board in the version one team room. This shows the list of sprint backlog items along with their progress.

Figure 1.4.1.2 Sprint Backlog (Source - Version one).
Burndown graphs are commonly used in Scrum projects to give the team an understanding of the amount of work remaining for the Sprint. As a team works together, it develops its own style of creating and maintaining the Sprint Backlog. It also demonstrates unique work patterns, some working consistently, some in bursts, some at the end of a Sprint. Some seek pressure, while others seek regularity. Across time, the backlog charts of each team develop predictable patterns. They stabilize as the team learns the technology, the business or product domain, and each other.

These chart patterns are called Sprint signatures (Kane Mar, 2006).

![Figure 1.4.1.3 Burndown Chart (Source - Version one)](image)

In Scrum, velocity (Beth Macy, 2013) is how much product backlog effort a team can handle in one sprint. This can be estimated by viewing previous sprints, assuming the team composition and sprint duration are kept constant. Velocity is used in sprint planning to fill a sprint.

Velocity can be used as a guide for removing obstacles and roadblocks, increasing training, enforcing a firewall, predicting releases, descoping functionality, resource planning, and increasing product owner involvement. Below figure shows the average velocity shown along with the planned velocity and a sprint burn down.
Product Increment

At the end of a Sprint the new Increment must be in a usable condition and meet the Scrum Team’s Definition of Done. In Scrum, the Development Team delivers each Sprint an Increment. The increment must consist of thoroughly tested code that has been built into an executable, and the user operation of the functionality is documented either in Help files or user documentation. These requirements are documented in the Definition of Done. If everything works fine and the Development Team has estimated well, the Increment includes all items, which were planned in the Sprint Backlog, tested and documented.
**Scrum framework**

- A product owner creates a prioritized wish list called a product backlog.
- During sprint planning, the team pulls a small chunk from the top of that wish list, a sprint backlog, and decides how to implement those pieces. They break the features required into manageable tasks.
- The team has a certain amount of time to complete its work called as a sprint and is usually two to four weeks duration. The team meets each day to assess its progress (daily Scrum).
- The Scrum Master keeps the team focused on its goal by shielding them from external factors.
• At the end of the sprint, the work should be potentially shippable: ready to hand to a customer or show to a stakeholder.
• The sprint ends with a sprint review and retrospective by means of meetings.
• As the next sprint begins, the team chooses another set of the product backlog and begins working again considering them as sprint backlog.

Figure: 1.4.1.6 Scrum framework.

1.4.2 EXTREME PROGRAMMING (XP)
Extreme Programming (Beck, Kent 1999) is a discipline of software development based on values of simplicity, communication, feedback, courage, and respect. It works by bringing the whole team together in the presence of simple practices, with enough feedback to enable the team to see where they are and to tune the practices to their unique situation.
Core Practices

Whole Team

All the contributors to an XP project sit together, members of one team. This team must include a business representative or the Customer, who provides the requirements, sets the priorities, and steers the project. It is best if the Customer or one of her aides is a real end user who knows the domain and what is needed (Ronald E. Jeffries 1999). The team will of course have programmers. The team may include testers, who help the Customer define the customer acceptance tests. Analysts may serve as helpers to the Customer, helping to define the requirements. There is commonly a coach, who helps the team keep on track, and facilitates the process. There may be a manager, providing resources, handling external communication, coordinating activities. None of these roles is necessarily the exclusive property of just one individual: Everyone on an XP team contributes in any way that they can. The best teams have no specialists, only general contributors with special skills.

Planning Game

XP planning addresses two key questions in software development: predicting what will be accomplished by the due date, and determining what to do next. The emphasis is on steering the project rather than on exact prediction of what will be needed and how long it will take. There are two key planning steps in XP, addressing these two questions.

Release Planning is a practice where the Customer presents the desired features to the programmers, and the programmers estimate their difficulty. With the cost estimates in hand, and with knowledge of the importance of the features, the Customer lays out a plan for the project. Initial release plans are necessarily imprecise: neither the priorities nor the estimates are truly solid, and until the team begins to work, we won’t know just how fast they will go.
Even the first release plan is accurate enough for decision making, however, and XP teams revise the release plan regularly.

Iteration Planning is the practice whereby the team is given direction every couple of weeks. XP teams build software in two-week “iterations”, delivering running useful software at the end of each iteration. During Iteration Planning, the Customer presents the features desired for the next two weeks. The programmers break them down into tasks, and estimate their cost (at a finer level of detail than in Release Planning). Based on the amount of work accomplished in the previous iteration, the team signs up for what will be undertaken in the current iteration.

These planning steps are very simple, yet they provide very good information and excellent steering control in the hands of the Customer. Every couple of weeks, the amount of progress is entirely visible. There is no “ninety percent done” in XP: a feature story was completed, or it was not. This focus on visibility results in a nice little paradox: on the one hand, with so much visibility, the Customer is in a position to cancel the project if progress is not sufficient. On the other hand, progress is so visible, and the ability to decide what will be done next is so complete, that XP projects tend to deliver more of what is needed, with less pressure and stress.

**Customer Tests**

As part of presenting each desired feature, the XP Customer defines one or more automated acceptance tests to show that the feature is working. The team builds these tests and uses them to prove to themselves, and to the customer, that the feature is implemented correctly. Automation is important because in the press of time, manual tests are skipped. That’s like turning off your lights when the night gets darkest (Ronald E. Jeffries 1999).

The best XP teams treat their customer tests the same way they do programmer tests: once the test runs, the team keeps it running correctly.
thereafter. This means that the system only improves, always notching forward, never backsliding.

**Small Releases**

XP teams practice small releases in two important ways:

First, the team releases running, tested software, delivering business value chosen by the Customer, every iteration. The Customer can use this software for any purpose, whether evaluation or even release to end users (highly recommended). The most important aspect is that the software is visible, and given to the customer, at the end of every iteration. This keeps everything open and tangible.

Second, XP teams release to their end users frequently as well. XP Web projects release as often as daily, in house projects monthly or more frequently. Even shrink-wrapped products are shipped as often as quarterly.

It may seem impossible to create good versions this often, but XP teams all over are doing it all the time. See Continuous Integration for more on this, and note that these frequent releases are kept reliable by XP’s obsession with testing, as described here in Customer Tests and Test-Driven Development.

**Simple Design**

XP teams build software to a simple but always adequate design. They start simple, and through programmer testing and design improvement, they keep it that way. An XP team keeps the design exactly suited for the current functionality of the system. There is no wasted motion, and the software is always ready for what’s next.

Design in XP is not a one-time thing, or an up-front thing, it is an all-the-time thing. There are design steps in release planning and iteration planning, plus
teams engage in quick design sessions and design revisions through refactoring, through the course of the entire project. In an incremental, iterative process like Extreme Programming, good design is essential. That’s why there is so much focus on design throughout the course of the entire development.

**Pair Programming**

All production software in XP is built by two programmers, sitting side by side, at the same machine. This practice ensures that all production code is reviewed by at least one other programmer, and results in better design, better testing, and better code.

It may seem inefficient to have two programmers doing “one programmer’s job”, but the reverse is true. Research into pair programming shows that pairing produces better code in about the same time as programmers working singly. That’s right: two heads really are better than one.

Some programmers object to pair programming without ever trying it. It does take some practice to do well, and you need to do it well for a few weeks to see the results. Ninety percent of programmers who learn pair programming prefer it, so we highly recommend it to all teams.

Pairing, in addition to providing better code and tests, also serves to communicate knowledge throughout the team. As pairs switch, everyone gets the benefits of everyone’s specialized knowledge. Programmers learn, their skills improve, and they become more valuable to the team and to the company. Pairing, even on its own outside of XP, is a big win for everyone.

Extreme Programming is obsessed with feedback, and in software development, good feedback requires good testing. Top XP teams practice “test-driven development”, working in very short cycles of adding a test, then making it work. Almost effortlessly, teams produce code with nearly 100
percent test coverage, which is a great step forward in most shops. (If your programmers are already doing even more sophisticated testing, more power to you. Keep it up, it can only help!).

It isn’t enough to write tests: you have to run them. Here, too, Extreme Programming is extreme. These “programmer tests”, or “unit tests” are all collected together, and every time any programmer releases any code to the repository (and pairs typically release twice a day or more), every single one of the programmer tests must run correctly. One hundred percent, all the time! This means that programmers get immediate feedback on how they’re doing. Additionally, these tests provide invaluable support as the software design is improved.

**Design Improvement**

Extreme Programming focuses on delivering business value in every iteration. To accomplish this over the course of the whole project, the software must be well-designed. The alternative would be to slow down and ultimately get stuck. So XP uses a process of continuous design improvement called Refactoring, from the title of Martin Fowler’s book, “Refactoring: Improving the Design of Existing Code”.

The refactoring process focuses on removal of duplication (a sure sign of poor design), and on increasing the “cohesion” of the code, while lowering the “coupling”. High cohesion and low coupling have been recognized as the hallmarks of well-designed code for at least thirty years. The result is that XP teams start with a good, simple design, and always have a good, simple design for the software. This lets them sustain their development speed, and in fact generally increase speed as the project goes forward.

Refactoring is, of course, strongly supported by comprehensive testing to be sure that as the design evolves, nothing is broken. Thus the customer tests
and programmer tests are a critical enabling factor. The XP practices support each other: they are stronger together than separately.

Continuous Integration

Extreme Programming teams keep the system fully integrated at all times. We say that daily builds are for wimps: XP teams build multiple times per day. (One XP team of forty people builds at least eight or ten times per day!)

The benefit of this practice can be seen by thinking back on projects you may have heard about (or even been a part of) where the build process was weekly or less frequently, and usually led to “integration hell”, where everything broke and no one knew why.

Infrequent integration leads to serious problems on a software project. First of all, although integration is critical to shipping good working code, the team is not practiced at it, and often it is delegated to people who are not familiar with the whole system. Second, infrequently integrated code is often — I would say usually — buggy code. Problems creep in at integration time that are not detected by any of the testing that takes place on an unintegrated system. Third, weak integration process leads to long code freezes. Code freezes mean that you have long time periods when the programmers could be working on important shippable features, but that those features must be held back. This weakens your position in the market, or with your end users.

Collective Code Ownership

On an Extreme Programming project, any pair of programmers can improve any code at any time. This means that all code gets the benefit of many people’s attention, which increases code quality and reduces defects. There is another important benefit as well: when code is owned by individuals, required features are often put in the wrong place, as one programmer discovers that he needs a feature somewhere in code that he does not own.
The owner is too busy to do it, so the programmer puts the feature in his own code, where it does not belong. This leads to ugly, hard-to-maintain code, full of duplication and with low (bad) cohesion.

Collective ownership could be a problem if people worked blindly on code they did not understand. XP avoids these problems through two key techniques: the programmer tests catch mistakes, and pair programming means that the best way to work on unfamiliar code is to pair with the expert. In addition to ensuring good modifications when needed, this practice spreads knowledge throughout the team.

**Coding Standard**

XP teams follow a common coding standard, so that all the code in the system looks as if it was written by a single — very competent — individual. The specifics of the standard are not important: what is important is that all the code looks familiar, in support of collective ownership.

**Metaphor**

Extreme Programming teams develop a common vision of how the program works, which we call the “metaphor”. At its best, the metaphor is a simple evocative description of how the program works, such as “this program works like a hive of bees, going out for pollen and bringing it back to the hive” as a description for an agent-based information retrieval system.

Sometimes a sufficiently poetic metaphor does not arise. In any case, with or without vivid imagery, XP teams use a common system of names to be sure that everyone understands how the system works and where to look to find the functionality you’re looking for, or to find the right place to put the functionality you’re about to add.
Sustainable Pace

Extreme Programming teams (Ronald E. Jeffries 1999) are in it for the long term. They work hard, and at a pace that can be sustained indefinitely. This means that they work overtime when it is effective, and that they normally work in such a way as to maximize productivity week in and week out. Extreme Programming is a discipline of software development based on values of simplicity, communication, feedback, and courage. It works by bringing the whole team together in the presence of simple practices, with enough feedback to enable the team to see where they are and to tune the practices to their unique situation.

1.4.3 OTHER METHODOLOGIES

Test-driven development (TDD)

Test-driven development (TDD) (Beck 2003; Astels 2003), is an evolutionary approach to development which combines test-first development where you write a test before you write just enough production code to fulfill that test and refactoring (Scottwambler, 2013). In other words, it’s one way to think through your requirements or design before your write your functional code (implying that TDD is both an important agile requirements and agile design technique). Another view is that TDD is a programming technique. As Ron Jeffries likes to say, the goal of TDD is to write clean code that works.

FDD

Feature-Driven Development (FDD) is a client-centric, architecture-centric, and pragmatic software process. The term "client" in FDD is used to represent what Agile Modeling (AM) refers to as project stakeholders or eXtreme Programming (XP) calls customers. FDD was first introduced to the world in 1999 (Scottwambler, 2013). As the name implies, features are an
important aspect of FDD. A feature is a small, client-valued function expressed in the form <action><result><object>. For example, "Calculate the total of a sale", "Validate the password of a user", and "Authorize the sales transaction of a customer". Features are to FDD as use cases are to the Rational Unified Process (RUP) and user stories are to Scrum – they are a primary source of requirements and the primary input into your planning efforts.

**DSDM**

DSDM (Dynamic Systems Development Method) is a robust agile project management and delivery framework that delivers the right solution at the right time (DSDM consortium n.d).

DSDM has been for many years the leading, proven agile approach, providing governance and rigor along with the agility and flexibility demanded by organizations today. The approach is the culmination of practitioners' experience drawn from a wide range of public and private sector projects over nearly two decades.

The DSDM Philosophy is that any project must be aligned to clearly defined strategic goals and focus upon early delivery of real benefits to the business. DSDM is vendor-independent, covers the entire lifecycle of a project and provides best practice guidance for on time, in budget delivery of projects – with proven scalability to address projects of all sizes and for any business sector.

DSDM advocates the use of several proven techniques, including

- Facilitated Workshops.
- Modelling and Iterative Development.
- MoSCoW Prioritization.
- Time boxing.
DSDM is designed to be easily tailored and used in conjunction with traditional methods such as PRINCE2 or to complement other agile approaches such as Scrum.

**Kanban**

The Kanban Method (David J. Anderson, 2010) is an approach to incremental, evolutionary process and systems change for organizations. Kanban is

- a way to organize the chaos that surrounds so many delivery teams by making the need for prioritization and focus clear.
- a way to uncover workflow and process problems so you may solve them in order to deliver more consistently to your client/customer/etc.

Kanban accomplishes these things by introducing constraints into the system to optimize the flow of value. Flow is king. Just remember that you want to focus on the flow of VALUE! If you have great flow but what’s being sent through isn’t valuable then your work has been for naught. If you can’t get your business value flowing out the door consistently, your business is not performing optimally. In addition to the above, by focusing on flow, Kanban resets your brain to value finishing over starting.

### 1.4.4 AGILE PRACTICES

Agile development is supported by a bundle of concrete practices suggested by the agile methods, covering areas like requirements, design, modeling, coding, testing, project management, process, quality, etc. Some notable agile practices include

- Backlogs (Product and Sprint).
- Behavior-driven development (BDD).
• Cross-functional team.
• Domain-driven design (DDD).
• Information radiators (Scrum board, Task board, Burn down chart).
• Iterative and incremental development (IID).
• Planning poker.
• Refactoring.
• Test-driven development (TDD).
• Agile testing.
• Time boxing.
• Use case.
• User story.
• Story-driven modeling.
• Retrospective.
• Velocity tracking.
• Acceptance test-driven development (ATDD).
• Continuous integration (CI).
• Scrum meetings (Sprint planning, Daily scrum, Sprint review and retrospective).
• Pair programming.
• Agile Modeling.

The Agile Alliance has provided a comprehensive online collection with a map guide to the applying agile practices. In the subsequent chapters, we propose and evaluate more practices to overcome certain crucial issues faced by the distributed agile teams. Though the target is the distributed agile teams, but these practices when implemented would also help the teams that are co-located.
Method tailoring

In the literature, different terms refer to the notion of method adaptation, including 'method tailoring', 'method fragment adaptation' and 'situational method engineering'. Method tailoring is defined as a process or capability in which human agents determine a system development approach for a specific project situation through responsive changes in, and dynamic interplays between contexts, intentions, and method fragments.

Potentially, almost all agile methods are suitable for method tailoring. Even the DSDM method is being used for this purpose and has been successfully tailored in a CMM context. UIP, Third place working are examples of method
tailoring. Situation-appropriateness can be considered as a distinguishing characteristic between agile methods and traditional software development methods, with the latter being relatively much more rigid and prescriptive. The practical implication is that agile methods allow project teams to adapt working practices according to the needs of individual projects. Practices are concrete activities and products that are part of a method framework. At a more extreme level, the philosophy behind the method, consisting of a number of principles, could be adapted (Aydin, 2004).

Extreme Programming (XP) makes the need for method adaptation explicit. One of the fundamental ideas of XP is that no one process fits every project, but rather that practices should be tailored to the needs of individual projects. Partial adoption of XP practices, as suggested by Beck, has been reported on several occasions. Mehdi Mirakhorli (2008) proposes a tailoring practice that provides a sufficient road-map and guidelines for adapting all the practices. RDP Practice is designed for customizing XP. This practice, first proposed as a long research work in the APSO workshop at the ICSE 2008 conference, is the proposed and applicable method for customizing XP. Although it is specifically a solution for XP, this practice has the capability of extending to other methodologies. At first glance, this practice seems to be in the category of static method adaptation but experiences with RDP Practice says that it can be treated like dynamic method adaptation. The distinction between static method adaptation and dynamic method adaptation is subtle.

1.4.5 THE SHIFT TO DISTRIBUTED AGILE TEAMS (DAT)

Since the emergence of agile methods in the mid-1990’s, the business of software development has changed dramatically. Today’s software development organizations are more likely to outsource at least part of their development. They are also more likely to expand their business into other countries—including high-growth countries like Russia and China. In other words, their business is more likely to be distributed. And with increasing
access to high-speed Internet service and improvements in distance collaboration tools, even employees within the same city are more likely to telecommute and work with their colleagues in a distributed fashion at least part of the time (Elizabeth Woodward, Steffan Surdek and Matthew Ganis, 2011).

**Globally Distributed Teams to Reduce Costs**

With a well thought-out plan to best leverage the talent in multiple countries, it can be less expensive to develop a product. Working with distributed teams where the talent is available to do the work can sometimes reduce labor and business operations costs.

**Reaching Market More Quickly with the “Follow the Sun” Model**

The team with members in the U.S. and China essentially has the possibility of working on a project 24 hours a day and making significantly more progress than a team working on a standard (8-hour) workday. By working with distributed teams, it is possible for companies to work a full 24-hour day and get their product to market more quickly.

**Acquisitions**

Another trend that increases team distribution is market consolidation, which results in an increase in acquisitions. These acquisitions are likely to result in distributed teams as the combined companies begin working together to integrate their products.
Expanding for Innovation and Thought Leadership

Another business trend that is increasing team distribution is the search for knowledge, intellectual property, and innovation. Studies have found that multinational companies produce more ideas than their purely domestic counterparts (Criscuolo 2005). Having more researchers and access to a larger “worldwide pool of information” fosters the generation of more ideas. In a knowledge-based economy, innovation and thought-leadership are critical to the success of a business. Hiring software development team members in other countries as part of the search for innovation naturally increases distribution of teams.

Telecommuting

Telecommuting is also a growing trend that increases the likelihood that even teams working in the same geographical areas will work as a distributed team. In a 2008 CompTIA study, 78 percent of the respondents reported that some employees within their organizations telecommute at least part-time (CompTIA 2008). Telecommuting can provide a significant increase in productivity, cost savings for companies, and a decrease in their carbon footprint. Employers are also using telecommuting as an incentive that attracts new employees who

Types of Distributed Teams That Have Emerged

As software development teams have become more distributed, they have adopted different patterns or models of distribution. Each model of distribution has a different set of challenges associated with it. Scrum Team members will recognize there are four general levels of challenges with each of the different Scrum activities. The subsequent chapters in this book provide suggestions
and recommendations for dealing with Scrum practices at each of the levels of distribution.

By decrease in the distribution, these levels are as follows:

1. Distributed with No Overlapping Work Hours
2. Distributed with Overlapping Work Hours
3. Collocated Part-Time
4. Collocated

**Co-location versus Distributed Teams**

Teams co-locate because it maximizes their ability to communicate in person. Working in the same room is core to all the agile methodologies, including scrum. As Ken Schwaber (2007), author of The Enterprise and Scrum, said, “The best communication is face to face, with communications occurring through facial expression, body language, intonation and words. When a white board is thrown in and the teams work out design as a group, the communication bandwidth absolutely sizzles.” Given that communication is such a significant part of the efforts involved in delivering software, the distributed teams are prevalent reflecting the reality of doing business today: a company’s need to have a global presence, to access global talent and to develop outsourced options. However, Some of the benefits of colocation are Questions gets answered quickly, Problems are fixed on the spot, Less friction occurs between interactions, Trust is gained and awarded much more quickly. These are harder to achieve in Distributed teams

**Challenges Faced by Distributed Teams**

Distributed teams experience the same problems as collocated teams, but in addition they have more issues as the team is not located at a single location.
The below are some of the issues faced in distributed teams and we would be dealing with them in detailed in the coming chapters.

- Communicating with Distributed Team Members
- Time Zones and Working Hours
- Cultural Differences
- Language Differences
- Communication gaps
- Infrastructure issues
- Mistrust in the team members
- Lack of face to face communication
- Less control on team
- Connectivity issues
- Roadblocks in learnings
- Difficult pairing

We discuss these issues in detailed in subsequent chapters and propose solution in the form of new practices to overcome certain crucial issues and evaluate them.

### 1.5 A SURVEY OF LITERATURE

Elizabeth Woodward’s *et al.* (2011) study on Challenges faced in Distributed teams is noteworthy and deals with issues faced in the teams that are distributed. The authors identifies and discusses on some of the challenges that distributed teams experience and also provides general tips for improving team communication regardless where the team is within the sprint. They also
investigates the role of the time zone differences, cultural differences, language differences that make the team communication difficult and then they suggest some tips to overcome them.

The author also cites the team dynamics, telephone dynamics and the issues faced by the teams and how they overcome it in the hundred plus distributed teams she has worked. Through many case studies, a set of lessons are prepared and these lessons can help the teams practicing distributed agile development process in their projects. To ensure successful execution of projects and timely delivery the experiences depicted in these lessons are very useful. The author while successfully pointing out the major hurdles on the distributed projects also gives various solutions to overcome them.

Lyssa Adkins (2012) suggests the agile mashups for better working of agile teams. The author noted that when teams mashed up scrum with something else, such as lean or user centered design, they enjoyed success if they kept scrum framework mostly intact and agile manifesto completely intact. It was also cited that whenever the team lets the scrum framework and agile manifesto fall by the wayside as they sought a way of working that fits them best, they often struggled and sometimes outright failed.

Roman Pitchler (2012) analysis on the agile product management with scrum and investigation on how to work, communicate with customers, users and other stakeholders, the common mistakes done by the agile teams do etc., is commendable. The author emphasizes that the product owner must be an effective communicator and negotiator and should align with different stakeholders to bridge the gap between the customers and the development teams.
A. Powell et al. (2004) findings on the contextual issues surrounding virtual teams provides a review of previously published work and reports on the findings from early virtual team research in an effort to take stock of the current state of the art. The review is organized around the input-process-output model and categorizes the literature into issues pertaining to inputs, socio-emotional processes, task processes, and outputs.

Saonee Sarker et al. (2001) mentions that study is done to provide a deeper understanding of agility through an intensive study of the distributed ISD experience in TECHCOM, an organization widely recognized for its excellence in IT development and use. The study reveals that agility should be viewed as a multifaceted concept having three dimensions: resource, process, and linkage. Resource agility is based on the distributed development team's access to necessary human and technological resources. Process agility pertains to the agility that originates in the team's systems development method guiding the project, its environmental scanning, and sense-making routines to anticipate possible crises, and its work practices enabling collaboration across time zones. Linkage agility arises from the nature of interactional relationships within the distributed team and with relevant project stakeholders, and is composed of cultural and communicative elements. The author highlights some of the difficulties in developing agility in distributed ISD settings, provides actionable tactics, and suggests contingencies wherein different facets of agility may become more (or less) critical.

One-Ki (Daniel) Lee et al. (2010) investigates on the risk management in globally distributed software development (GDSD) projects and observes that it is becoming a critical area of concern for practitioners. The risks in GDSD projects can be dynamic due to the multiplicity in various aspects of GDSD projects (e.g., multi-locations, multi-cultures, multi-groups, multi-standards,
and multi-technologies). This multiplicity nature leads to dynamic interactions among the internal (i.e., people, process, and technology) and external elements of a GDSD project. This study aims to develop a new framework to identify the dynamic risks in GDSD projects and mitigate them using agile risk management practices.

J. Kotlarsky (2005) focuses his attention on how the Global distribution of software development has become widespread over the last decade in his study. The author says that there are a number of economic and technical drivers that are likely to further accelerate the growth of distributed software development. For economic and financial considerations, many companies are switching to globally distributed development and/or offshore outsourcing of products and services. He also proposes how the software industry has started to adopt a more modular or Component-Based (CB) architecture that facilitates development of Software products with a long lifetime. In terms of system structure, CB system architecture is considered to be a key success of systems with a long life cycle (Crnkovic and Larsson 2002). As compared to a monolithic software system, a CB system is considered to be flexible, extensible, and reusable (Crnkovic and Larsson 2002). Furthermore, a CB system is easier and more effective to maintain, because it can be maintained in parts (by components), as opposed to a monolithic system that needs to be maintained as a whole (Verbraeck et al., 2002).

O.K. D. Lee et al. (2006) in a study discusses about the Agile IT strategy, infrastructure, and project management and the role played by them as key elements for realizing agility in Global distribution of software development projects. The author proposes on how to align IT components to achieve agility in globally distributed system development.
Constance A. Steinkuehler, Dmitri Williams (2006) made an extensive study to examine the form and function of massively multiplayer online games (MMOs) in terms of social engagement. Combining conclusions from media effects research informed by the communication effects literature with those from ethnographic research informed by a sociocultural perspective on cognition and learning, they have presented a shared theoretical framework for understanding the extent to which such virtual worlds are structurally similar to “third places” (Oldenburg, 1999) for informal sociability, and their potential function in terms of social capital (Coleman, 1988; Putnam, 2000). The authors conclusion is that by providing spaces for social interaction and relationships beyond the workplace and home, MMOs have the capacity to function as one form of a new “third place” for informal sociability. This extensive study and results produced proves that the Participation in such virtual “third places” appears particularly well suited to the formation of bridging social capital—social relationships that, while not usually providing deep emotional support, typically function to expose the individual to a diversity of worldviews. This would mean that the third places would also suit the teams who are isolated from the teams and are located at distance places without much social interactions and emotional support.

Y Marco R. Della Cava (2006) studied and presented the facts of working out of third place. The author states that Working from a place like thirdplace is less stressful than being in an office, and finds that it caters to get a lot more done. He also quotes that an estimated 30 million Americans, or roughly one-fifth of the nation's workforce, are part of the employees who spend significant hours each month working outside of a traditional office. This study gives an insight of how the thirdplace working is gaining popularity day by day and how to use it for the betterment of the teams productivity while the team being in a happy work culture.
In a potential study about the alternative work places in an article ‘The new oases’ in The Economist (2008), the academic name for such spaces is termed as “third places”, a term originally coined by the sociologist Ray Oldenburg in his 1989 book, “The Great, Good Place”. At the time, long before mobile technologies became widespread, Oldenburg wanted to distinguish between the sociological functions of people’s first places, their second places and the public spaces that serve as safe, neutral and informal meeting points. As Oldenburg (1989) saw it, a good third place makes admission free or cheap—the price of a cup of coffee, say—offers creature comforts, is within walking distance for a particular neighborhood and draws a group of regulars. The eponymous bar in the television series “Cheers”, “where everybody knows your name”, is an example.

Oldenburg’s (1989) thesis was that third places were in general decline. More and more people, especially in suburban societies such as America’s, were moving only between their first and second places, making extra stops only at alienating and anonymous locations such as malls, which in Oldenburg’s opinion fail as third places. Society, Oldenburg (1989) feared, was at risk of coming unstuck without these venues for spreading ideas and forming bonds. No sooner was the term coined than big business queued up to claim that it was building new third places. The most prominent was Starbucks, a chain of coffee houses that started in Seattle and is now hard to avoid anywhere. Starbucks admits that as it went global it lost its ambiance of a “home away from home”. However, it has also spotted a new opportunity in catering to nomads. Its branches offer not only sofas but also desks with convenient electricity sockets. These days Starbucks makes bigger news when it switches Wi-Fi providers—it dropped T-Mobile for AT&T in February—than when it sells a new type of coffee bean. Bookshops such as Barnes & Noble are also offering “more coffee and crumbs”, as Oldenburg puts it, as are churches, YMCAs and public libraries.
James Katz (2008) at Rutgers observes that cyber-nomads are “hollowing them out”. It is becoming commonplace for a café to be full of people with headphones on, speaking on their mobile phones or laptops and hacking away at their keyboards, more engaged with their e-mail inbox than with the people touching their elbows. These places are “physically inhabited but psychologically evacuated”, says Katz, which leaves people feeling “more isolated than they would be if the café were merely empty”. That is because the “physical presence of other human beings is psychologically and neurologically arousing” but now produces no reward. Quite simply, he says, we have not evolved biologically to be happy in these situations.

Jim Highsmith, Alistair Cockburn (2001) does an in depth analysis and discussion about the team proximity and intense interaction between team members and how they are hallmarks of all agile methods. The authors also points out how XP is noted for pair programming, although the practice has been around for years under other names. Crystal, Scrum, and ASD advocate close collaboration practices including barrier-free collocated teams. Lean Development stresses team interaction. Using agile development methods requires close customer partnerships. If the customers, either internal department representatives or marketing product managers, don’t have a good sense of direction and wander around in strange patterns, agile developers will follow them (with occasional admonitions, of course). Poor customers result in poor systems.

Mark Lycett et al. (2003) of Brunel University thoroughly discusses about the situated process, quality frameworks and the way they offer to resolve the tensions that arise when introducing agile methods into standardized software development engineering. The authors notes that the agility strongly favors human communication and collaboration over defined and repeatable activities as mechanisms for developing quality software. The study focuses on the difficulties that many organizations face with the agile approach. First, in the cultural change necessary to convince management of its benefits and,
second, in its feasibility within the increasingly global market forces, regulatory practices, and government oversight that fuel standardization. The authors further discuss about the process theories and realities, situated process, situated quality and resolving tensions

Mira Kajko-Mattsson (2008) in his work studies and reports the agile evolution and maintenance trenches on the lifecycle problems related to system and process documentation. The report gives an indication of the problems that have been encountered within 18 agile organizations and the practices followed to overcome them.

M.Pikkarainen et al. (2008) presented the results from a study which examined the impact of XP and SCRUM practices on communication within software development teams and within the focal organization. The research was carried out as a case study in F-Secure where two agile software development projects were compared from the communication perspective. The study shows that agile practices improve both informal and formal communication. However, it further indicates that, in larger development situations involving multiple external stakeholders, a mismatch of adequate communication mechanisms can sometimes even hinder the communication. The study highlights the fact that hurdles and improvements in the communication process can both affect the feature requirements and task subtask dependencies as described in coordination theory

Siva Dorairaj et al. (2010) in their extensive study outlines the proposed research on distributed agile projects and it explores distributed agile projects using the grounded theory research methodology, understand the challenges faced by the agile practitioners in distributed projects, factors that should be
present for successful execution and collates the strategies adhered by agile practitioners to effectively manage distributed projects

Bhushan Lal Sahu, Rajesh Tiwari (2012) provide a comprehensive study on the motivation factors of adopting cloud computing, review the several cloud deployment and service models. The authors also explore certain benefits of cloud computing over traditional IT service environment-including scalability, flexibility, reduced capital and higher resource utilization which are considered as adoption reasons for cloud computing environment. They also observes that security, privacy, internet dependency and availability as avoidance issues and concludes that Cloud computing have several benefits over traditional (non-cloud) environment and have capability to handle most sudden, temporary peaks in application demand on cloud infrastructures.

M Moniruzzaman & Dr. Syed Akhter Hossain (2013) significantly identifies and describes the major factors, that agile development approach improves software development process to meet the rapid changing business environments. The Authors also provide a brief comparison of agile development methodologies with traditional systems development methodologies, and discuss current state of adopting agile methodologies. It is speculated that from the need to satisfy the customer through early and continuous delivery of the valuable software, agile software development is emerged as an alternative to traditional plan-based software development methods. In total, the authors through this work provides an in-depth understanding, the major benefits of agile development approach to software development industry, as well as provide a comparison study report of ASDM over TSDM.
Scott Ocamb (2013) discusses about the agile manifesto which was authored by seventeen independent-minded practitioners at a ski lodge in Utah in 2001. He re-iterates the core that these people had in mind and how the allowance of flexibility would increase our chances of success. The Author Scott Ocamb (2013) in a separate article discusses about the 12 agile manifesto principles followed by four values of agile manifesto and they were also discussed by Paul I. Pazderski (2010).

(Beth Macy, 2013) focuses on defining the velocity and how reliable is the velocity during the sprint planning and how to use it for the better planning of the sprint.

(Kane Mar, 2006) discusses about the ways to find out what the projects burndown signature says about your team. This article is a collection of 7 burndown signatures that seem to be the most common – the fakey fakey, the late learner, the middle learner, the early learner, the never-never, the scope increase.

Ken Schwaber (2007) discusses thoroughly the change management for organizational and interpersonal processes, explaining how to successfully adopt Scrum across entire organization. As a cofounder of Scrum, Ken, draws from decades of experience, answering the questions through case studies of proven practices and processes. With these case studies, it’s easy to learn adopting and adapting to scrum in the enterprise and gain profound levels of transparency into the development.

Alexandre Magno (2009) in his study highlights that Scrum, like most of the agile processes, was created with the software projects pain points in mind.
However, those pain points are not exclusive to the software projects, and they are present in whatever projects where change is a constant. The author discussed that in the executive business world today, change is a constant, and to be prepared to them is a must. The author also provides his experiences on using Scrum with senior management and how the results supported Scrum promotion for other projects inside the company.

Hall E (1998) in his extensive study says that risk is inherent in the development of any large software system. Leading software companies use quantitative risk management methods as a more useful approach to achieve success. The author aims the study and results for busy professionals charged with delivering high-quality products on time and within budget. Managing Risk is a comprehensive guide that describes a success formula for managing software risk. He describes a risk management road map designed to take the situation of crisis to control of the software projects.

A.Cockburn,Williams L (2003) in their article thoroughly discusses about the agile software development methods and the way how the software development is more empirical when compared with the manufacturing processes. The authors further provides valuable observations on agile versus plan driven models.

Helen Sharp et al. (2008) discuss on the ways in which Agile software development promotes feedback, discipline and close collaboration between all members of the development team, and de-emphasizes documentation, ‘big design up front’ and hierarchical processes. Agile teams tend to be co-located and multi-disciplinary, and rely heavily on face-to-face communication and seemingly simple physical artefacts to support interaction. The authors focus on the functionality of two key physical artefacts – the story card and
the Wall – which, individually and in combination, underpin the team’s activity. These artefacts have two main roles – one which enables a shared understanding of requirements and one which facilitates the development process itself. They considered these roles from two perspectives: a notational perspective and a social perspective. This discussion shows how the two perspectives – the notational and the social – intertwine and are mutually supportive. The authors says any attempt to replace these physical artefacts with alternative support for an agile team needs to take account of both perspectives, and the complex relationships between them.

Sallyann Bryant et al. (2007) attempts to unpick the pair programming process through the analysis of verbalizations from a number of commercial studies. The authors focus particularly on the roles of the two programmers and what their key characteristics and behaviors might be and also suggest that a key factor in the success of pair programming could be the associated increase in talk at an intermediate level of abstraction.

M. Angioni et al. (2008) intensely discuss about the Extreme Programming (XP), an Agile Methodology (AM). The authors provides many reasons leading a XP team to adopt Web based tools to support XP practices. The way of how the full Object Oriented implementation providing a powerful support for extracting all data represented in the model that the system implements is discussed.

Carlos Ferreira, Jason Cohen (2008) in their work develops and tests a research model that hypothesizes the effects of five characteristics of agile systems development - iterative development; continuous integration; test-driven design; feedback; and collective ownership, on two dependent stakeholder satisfaction measures, namely stakeholder satisfaction with the development process and with the development outcome. An empirical study
of 59 South African development projects (using self-reported data) provided support for all hypothesized relationships and generally supports the efficacy of Agile methodologies. Iteration and integration together with collective ownership have the strongest effects on the dependent satisfaction measures.

Kieran Conboy, Brian Fitzgerald (2010) primary motivation in their study was to investigate the method tailoring topic in more depth. The authors considered the tailoring issue from two complementary perspectives—that of the characteristics of the actual method being tailored, and also the characteristics of individual developers responsible for method tailoring.