Chapter 2

Literature Review

Global sourcing and distributed development of software applications have become a business reality in IT industry. Organizations adopt the practice of distributed agile in order to reap the benefits of both agile and distributed development. Distributed development and agile approach differ significantly in their key tenets and hence combining the two methods pose significant challenges to software projects. Hence, in this chapter we lay the foundation of the key aspects of software development in distributed agile environment. It starts with describing the traditional approach for software development and the evolution of agile development approach in section 2.2. It introduces the concepts of distributed software development (DSD) in section 2.3 and how agile is being blended with DSD to create cost-effective solutions to meet the changing business needs in section 2.4. Further, the benefits and challenges of using Distributed Agile Development (DAD) are being discussed. The concepts related to risk management are being discussed in section 2.5 and the theoretical framework which forms the basis for the construction of Risk Management Framework for DAD projects is presented in the section 2.6 of this chapter. The related areas of research and the corresponding studies are consolidated in section 2.7.

2.1 Traditional Software Development

Software development is growing rapidly due to the fact that software has merged into many diverse fields and is becoming increasingly complex (Hneif & Hockow, 2009). Over last few years, software has become an important component of every business for achieving success.

Software engineering involves activities that are focused on solving problems of immense complexity (Royce, 1997). Software entities are more complex than any other human construct because no two parts are alike. In this respect, software system differs profoundly from computers, buildings, or automobiles, where there are repeated elements. Complexity further leads to miscommunication amongst the team members, product flaws, cost overruns and schedule delays.
To fulfill the need to develop increasingly complex software systems, software development approaches have evolved during the time. In the 1970’s, to overcome the problems encountered in managing large custom software development projects, the widely known “waterfall” model was introduced (Royce, 1970). The waterfall model proved to be successful for early problems, but it also had some pitfalls like poor handling of requirements and lack of customer involvement (Guntamukkala, Wen & Tarn, 2006). In 1980’s, models which could deliver “increments” of the software, referred to as incremental models like Rapid Application Development (RAD) were introduced. These models combined the elements of the waterfall model applied in an incremental fashion. They focused on delivery of an operational product in each increment. Later a series of process models were introduced which could cater to the need of accommodating a product, which can evolve over time. These included the evolutionary models like prototyping, spiral model, which enabled the software developers to create more complete versions of the software (Pressman, 2005).

However, as software development involved more critical and dynamic industrial projects, new difficulties emerged according to the growth of companies. Problems like changing requirements, lack of customer involvement, low budgets, tight deadline and miscommunication were still not completely addressed (Hneif & Hockow, 2009). Hence, a new set of lightweight methodologies under the umbrella of ‘Agile Development’ joined the existing flexible models of software development. Agile software development refers to a group of software development methodologies aiming to more nimble and lighter development processes, making them more responsive to change (Lehtonen, 2009).

Agile methods have proved to be ten times more productive than traditional development models, though this has been achieved in the co-located teams (Phalnikar, Deshpande & Joshi, 2009). This methodology helps in overcoming the problems like frequent changes and leads to faster development and greater user satisfaction.

2.2 Agile Software Development
Agile Methods are a reaction to traditional ways of developing software and acknowledge the need for an alternative to documentation driven; heavyweight software development processes
(Beck et al., 2001). In the implementation of traditional methods, work begins with the elicitation and documentation of a “complete” set of requirements, followed by architectural and high-level design, development, and inspection. Agile approach, on the other hand strips away much of the heaviness associated with traditional methods in order to promote quick response to the changing environment and changing user needs (Erickson, Lyytinen, & Siau, 2005). Agile methods are actually a collection of different techniques (or practices) that share the same values and basic principles. Many are, for example, based on iterative enhancement, a technique that was introduced in 1975 (Basili & Turner, 1975).

Iterative development is an approach to build software (or anything) in which the overall lifecycle is composed of several iterations in sequence. Each iteration is a self contained mini project composed of activities such as requirement analysis, design, programming and test. The goal is to create a stable, integrated and tested, partially complete system known as iteration release, at the end of each iteration. This partial system grows incrementally with new features, iteration by iteration, in other words incremental development. This concept of ‘iterative and incremental’ approach or simply ‘iterative approach’ is the core of all agile methods. Hence, short time boxed iterations with adaptive, evolutionary refinement of plans and goals are basic practice shared by various agile methods. In addition, they promote practices that encourage simplicity, lightness, communication, self directed teams, programming over documentation (Larman, 2012).

2.2.1 Agile Software Development - Values and Principles

In 2001, a group interested in iterative, agile methods met to find a common ground. They came up with a manifesto and statement of principles. All the agile methods follow these four values and twelve principles. The practices in agile development reflect these principles and values. The focal values honored by the agilest are presented as follows:

- **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 (Beck et al., 2001).

The first value says that agile approach emphasizes on the relationships and communicability of software developers and the human role reflected in the contracts, as opposed to institutionalized processes and development tools.

The second value stresses on the releasing tested working software at frequent intervals. The developers are encouraged to create technically advanced, but simple codes, hence reduce the burden of documentation to an appropriate level.

According to the third value, although it is necessary to create well drafted contracts, but relationship between the developers and the client is more important. The negotiations between the client and the developers help to build and maintain viable relationship.

The fourth value says, that the development group, which comprise of both developers and the customers, should be well-informed, competent and authorized to consider possible adjustments which emerges during the development process. This means that the participants are prepared to make the changes and that also the existing contracts are formed with tools that support and allow these enhancements to be made (Abrahamsson, Salo, Ronkainen, & Warsta, 2002).

The twelve principles as stated in the Agile Manifesto are as follows (Beck et al., 2001):

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a
development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users
should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity—the art of maximizing the amount of work not done—is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and
adjusts its behavior accordingly.

**2.2.2 Overview of Specific Agile Methods**

Agile methods are based on the best practices of Japanese industry, particularly lean
development principles which were implemented by companies like Toyota and Honda and
knowledge management principles developed by Taekeuchi and Nonaka (2004). Under the
umbrella of agile development, there are a group of methods that are built on the idea of
flexibility and adaptability (Abrahamson et al., 2003; Meso & Jain, 2006). Hence, a brief
description of the agile methods which are prominently known is described below:

1. Adaptive Software Development (ASD) (Highsmith, 1999) stresses on incremental and
iterative development with continuous prototyping. Its emphasis is on the production of
high-value results derived from rapid adaptation to both external and internal events. In
ASD more importance is placed on adaptation rather than optimization.

2. Scrum (Schwaber & Beedle, 2002) is one of the more widely used agile methods, is a
team-based method for the management of development in volatile environments. It
emphasizes flexibility, adaptability, and productivity from an empirical vantage point.
Scrum adopts the iterative, incremental approach in order to respond to rapidly changing
requirements. As opposed to other methodologies, it is scalable and leaves some
flexibility in decisions regarding the implementation process. Scrum projects are split in
small time-boxed iterations, called sprints, each of length 1 to 6 weeks. Each sprint
consists of pre-sprint planning and post-sprint planning. Scrum works on the principles of
delivering frequent builds which are developed by small teams that maximize
communication and sharing of tacit and informal knowledge (Cohen, Lindval & Costa,
2003).

3. Extreme Programming (XP) (Beck, 2005) is a lightweight process which attempts to
address the constraints in software development and focuses on the ability to adapt to
rapidly changing requirements. XP revolves around twelve key principles which include:
- system metaphor
- planning game
- small releases
- simple design
- testing
- refactoring
- pair programming
- collective ownership
- continuous integration
- forty-hour week
- on-site customer
- coding standards

XP has become one of the most widely used agile methods and the subject of an increasing amount of research (e.g. Abrahamson et al., 2003; Abrahamsson & Koskel, 2004; Alshayeb & Li, 2005; Fruhling & Vreede, 2006).

4. Feature-Driven Development (FDD) (Palmer & Felsing, 2002) emphasizes the design
and building phases. It is very much iterative in nature, and focuses on generating
working results every two weeks. Through the frequent delivery of tangible results risk is
reduced and better quality can be ensured.

5. Agile Modeling (AM) (Ambler, 2002) focuses on modeling practices and cultural
principles. The models should be advanced enough to support both the design needs and
purpose of the documentation. One aim of AM is to limit the amount of models and
documentation. Clear communication and the organization of team structure are
emphasized to address the cultural issues that may arise throughout the development
process.

6. The Crystal Family (Cockburn, 1998) provides various methods which may be selected
based on the particular project. It also includes principles by which the methods may be
tailored to accommodate varying circumstances. Within the Crystal Family, a color
coding scheme indicates the heaviness of the method. Therefore, the appropriate-colored
method can be selected based on the size and critical nature of the project. This
combination of methodologies is flexible enough to integrate the practices of other agile methods such as XP and Scrum.

7. Dynamic Systems Development Method (DSDM) (Stapleton, 2003) was first released in 1994. It is based on the concepts of rapid application development. The overall idea is to establish the amount of time and resources that are available and then determine the amount of system functionality based on these parameters. This is in contrast to the traditional approach of adjusting time and resources until the necessary functionality is reached.

Although numerous methods have been identified, two concepts lie at the heart of each of these methods: working code and effective people. Only through creativity, team work, customer participation, and continuous feedback can effective projects be completed on time and within budget (Highsmith & Cockburn, 2001).

2.2.3 Explanation of the most widely used Agile Methods

Amongst all the agile methods, SCRUM and XP are the agile methods that have received maximum acceptability in the software industry (VersionOne, 2012), (VersionOne, 2013). Moreover, the findings of this study are also primarily based on the practices of SCRUM and XP, since the majority of projects whose data has been used were developed by using Scrum, Xp or variants of Scrum and Xp. Overall, Scrum and Xp are the two agile methods that are most frequently used by the practitioners. Hence, we consider it appropriate to describe some important concepts related to these two methods briefly in this section.

2.2.3.1 SCRUM – A Brief Summary

Scrumb is an iterative, incremental framework for projects and product or application development. The development occurs in time boxed, short cycles, no more than one month, called sprints. At the beginning of the sprint, a cross-functional team selects items (customer requirements) from a set of prioritized items. The team commits to complete the set of items during the sprint. Every day the team meets in order to understand the status of the project and adjusts its next steps accordingly. At the end of the sprint, the sprint demonstrates the working
software built during the sprint to the stakeholders. Based on the feedback obtained during the sprint review, the next stages of development are planned. SCRUM is based on the theme ‘inspect and adapt’, wherein, during the small development steps, the team inspects the product that is being developed and the efficiency of the process being used. It adapts the process and the practices used in order to achieve the business goals.

Scrum has three roles, namely, The Product Owner, The Team and The ScrumMaster, which together form a Scrum Team. The **Product Owner** is responsible for maximizing the Return on Investment (ROI) by identifying the product features, translating these into prioritized list, deciding which set of features should be developed in the next sprint and continually re-prioritizing and refining the list. In Scrum, there is one and only one person who serves as the Product Owner, who has the final authority and is ultimately responsible for the value of the work. The **Team** builds the product based on the inputs given by the Product Owner. Scrum team is “cross-functional”, which means that it includes all the expertise necessary to deliver a potentially shippable product at the end of each sprint. The team possess a unique characteristic of being “self-organizing” (or self-managing), which means that the Team decides what to commit to and how to deliver. It has a high degree of autonomy and accountability. The team in Scrum is seven plus minus two people and for software product the team includes people with skill in analysis, development, testing, interface design, database design, architecture, documentation etc. Application groups with many people are organized into multiple Scrum teams, each focused on different features for the product, and work in close collaboration. Every team does all the work including planning, analysis, programming and testing for the features to be developed, and hence are also known as feature teams (Deemer, Benefield, Larman, & Vodde, 2009).

The **ScrumMaster** helps the team members to learn and apply Scrum to achieve business value. The ScrumMaster is not the manager of the Team or the project manager; instead, the ScrumMaster serves the Team, protects them from outside interference, and educates and guides the Product Owner and the Team in the skillful use of Scrum.
The first step in Scrum requires the Product Owner to articulate the product vision which eventually evolves into a refined and prioritized list of features called the Product Backlog. The backlog provides a single view of everything that needs to be done by the team, in the order of priority. The list of items in the product backlog usually exists in the form of ‘user stories’. The product backlog is continuously updated and re-prioritized by the Product Owner to reflect the changes in the needs of the customer and moves by the competition. Over time, the Team tracks how much work on an average can be done in a Sprint which is referred to as the “Velocity”. Velocity is expressed in the same units as the Product Backlog item size estimates.

At the beginning of the Sprint, the Sprint Planning Meeting takes place which is divided in two distinct sub-meetings, namely, Sprint Planning Part One and Sprint Planning Part Two. During the Sprint Planning Part One, the Product Owner and the team review the high Priority items in the Product Backlog that needs to be implemented. The Sprint Planning Part Two focuses on detailed task planning for how to implement the items that the Team decides to take on. The Team selects the items from the Product Backlog and they commit to complete them by the end of the sprint. The items selected are broken into individual tasks, which are recorded in a document called the “Sprint Backlog.” The sprint planning meeting often last for number of hours, but no more than eight hours for a four–week Sprint (Deemer, Benefield, Larman, & Vodde, 2009).

Once the sprint starts, the Team engages in the development and testing activities. It performs a “Daily Scrum”, which is a 15 minute meeting, happening everyday at an appointed time. In the Daily Scrum, one by one, each member of the team reports three things to the other members of the team: (1) What they are able to get done since the last meeting (2) What they are planning to finish by the next meeting (3) Any blocks or impediments that are in the way. Usually, if any discussions are required, it is carried during the follow-up meeting, after which the team again adapts its practices to achieve its goals (Deemer, Benefield, Larman, & Vodde, 2009).

Since, the Team is self-managing; they must know how they are progressing through the sprint. They update their estimate of the amount of time remaining to complete their current task in the Sprint Backlog by creating Sprint Burndown Chart. The graph shows, each day, a new estimate
of how much work (measured in hours) remain until the team tasks are finished. Ideally, this graphs should have a downward sloping graph indicating “zero effort remaining” by the last day. After the sprint ends, there is a Sprint Review meeting, where the Team and the Product owner review the Sprint. The Sprint Review is an ‘inspect and adapt’ activity of the product by having a ‘demo’ of the product built by the Team and in-depth conversation between the Team and the Product Owner. Further, it is the responsibility of the ScrumMaster to ensure that the items being demonstrated as a part of the sprint review are ‘done’ according to the “Definition of Done” defined for the product or release by the Team. The Sprint Review is followed by a “Sprint Retrospective” which gives the Team and opportunity to discuss on which practices are working well and what’s not working, and agree to make some changes in the execution of the next sprint (Deemer et al., 2009).

After the Sprint review, the backlog is updated by the Product Owner and the next sprint cycle begins. There is no down time between the sprints which supports the agile principles of “sustainable pace” which says that the Team must work on regular hours at a reasonable level so that they can continue this cycle indefinitely. The key practices, artifacts and events of the Scrum are depicted in Figure 2.1.
2.2.3.2 Extreme Programming – A Brief Summary

One of the widely used agile methods for software development is Extreme Programming (XP), which promotes the development of rigorously tested code in small iterations by working in close collaboration with the end-user (Beck, 2000). XP emphasizes on keeping the team small and the code simple (Beck & Fowler, 2001). It encourages continual communications with customers and teams, maintaining simplicity, providing frequent feedback via testing, and dealing with problems proactively (Paulk, 2001). XP recommends the use of 12 practices for software development which are listed as follows (Beck, 1999):

**Planning game**: Customers decide the scope and timing of releases based on estimates provided by programmers. Programmers implement only the functionality demanded by the stories in this iteration.

**Small releases**: The system is put into production in a few months, before solving the whole problem. New releases are made often—anywhere from daily to monthly.
**Metaphor:** The shape of the system is defined by a metaphor or set of metaphors shared between the customer and programmers.

**Simple design:** At every moment, the design runs all the tests, communicates everything the programmers want to communicate, contains no duplicate code, and has the fewest possible classes and methods.

**Tests:** Programmers write unit tests minute by minute. These tests are collected and they must all run correctly. Customers write functional tests for the stories in an iteration. These tests should also all run though practically speaking, sometimes a business decision must be made comparing the cost of shipping a known defect and the cost of delay.

**Refactoring:** The design of the system is evolved through transformations of the existing design that keep all the tests running.

**Pair programming:** All production code is written by two people at one screen/keyboard/mouse.

**Continuous integration:** New code is integrated with the current system after no more than a few hours. When integrating, the system is built from scratch and all tests must pass or the changes are discarded.

**Collective ownership:** Every programmer improves any code anywhere in the system at any time if they see the opportunity.

**On-site customer:** A customer sits with the team full-time.

**40-hour weeks:** No one can work a second consecutive week of overtime. Even isolated overtime used too frequently is a sign of deeper problems that must be addressed.

**Open workspace:** The team works in a large room with small cubicles around the periphery. Pair programmers work on computers set up in the center.

The XP development cycle involves customer picking up the most valuable features, which are known as ‘user stories’. Based on the cost of the stories and the team’s speed; the team decides the set of stories to be developed in an iteration. The stories are converted to fine-grained tasks, which is taken up by individual programmer. The tasks are then converted into set of test cases that will demonstrate that the tasks are finished. The team emphasized on simplest possible design and continuously refactors the code to create a maintainable system (Beck, 1999).
2.3 Distributed Software Development (DSD)

In order to build large and complex systems, there was a need to get an access to a talented pool of resources. Moreover, it was required to create cost effective solutions and reduce the time required for the product to reach the market (Herbsleb & Moitra, 2001; Sengupta, Chandra & Sinha, 2006; Krishna, Sahay & Walshman, 2004). Many companies started distributed development to benefit from cheaper, faster and better development of software systems, products and services (Šmite, Wohlin, Gorschek & Feldt, 2009). Distributed development also minimized risk in case of natural catastrophes and other events (Lehtonen, 2009).

Distributed Software Development (DSD) is defined by Prikladnicki, as a software development process where at least one involved actors (project team, or customer or user), is physically distant from the others (Prikladnicki, Audy, & Evaristo, 2003). Another definition is given by Mohtashmi and co-authors as “Development of software with teams working at different, distributed sites, often with same specialties and background with infrequent face-to-face communication” (Mohtashmi, Marlowe, Kirova & Deek, 2006). DSD and its different forms are pictorially represented in Figure 2.2.

Brian Lings has categorized distributed development in intra-continental development as one category and inter-continental as another category also named as Global Software Development (GSD) (Lings, Lundell, Agerfalk & Fitzgerald, 2006). GSD can be considered as an extension of DSD. It possesses characteristics of DSD, like development being carried at geographically apart locations with team relying more on formal communication. Along with this, it possesses additional characteristics related to time-zone differences and socio-cultured distance.

Besides GSD, there are other related terms which are used in the literature including Virtual Teams, Offshore Outsourcing, Multisite Development, Dispersed Development, Offshoring, and Outsourcing.

These terms are briefly explained as:

- **Multisite Development**: A relatively self contained team working on sub-system or relatively well-decoupled part of the whole (Cockburn, 2002, p.81)
- **Dispersed Development:** Individual developers working in different locations.
- **Outsourcing:** Work is sub-contracted to other individuals or organizations. This does not always equate to distribution (Brown, 2007).
- **Off shoring:** work done in a lower cost country normally located some distance away from the country.
- **Near shoring:** work done in a neighboring low cost country.

In this study we are using ‘Distributed Software development (DSD)’ since it is a general term to represent software development with dispersed teams. DSD has been used in terms to represent the dispersion of software development teams at any scale and doesn’t have the connotations of great geographical distances found in other designations (Layman, Williams, Damian & Bures, 2006, p.781; Komi-Sirviö & Tihinen, 2006, p.765).

![Figure 2.2: Taxonomy for Distributed Software Development](Source: Pickladnicki et al., 2003)
Distributed Software Development is characterized by

- **Spatial distance**: This is the directional measure of the effort required for one actor to visit another at the latter’s home site. In general, low geographic distance is best measured in ease of relocating rather than in kilometers (Lings, Lundell, Agerfalk & Fitzgerald, 2006). In case of software development organizations, which develop distributed development projects, it refers to the distance of developers from each other and from customers or end-users. Prikladnicki, Audy, & Evaristo (2003) have developed a detailed typology for physical distribution among users, customers, and project teams, as well as within project teams, which is shown in Figure 2.2.

- **Time-zone distance**: Is a directional measure of the dislocation in time experienced by two actors wishing to interact. Temporal distance can be caused by time zone difference or time shifting work patterns. In general, low temporal distance improves opportunities for timely synchronous communication but may reduce management options. Time zone to a large extent is a confounding factor with distance (Lings et al., 2006).

- **Socio-cultural distance**: Is a directional measure of an actor's understanding of another actor's values and normative practices. It is a complex dimension, involving organizational culture, national culture and language, politics, and individual motivations and work ethics (Lings et al., 2006; Carmel, 1999).

Since teams practicing global software development are characterized by spatial, temporal and cultural distance, they have to face insurmountable challenges. Distance negatively impacts the coordination and control, which are necessary for an organizational unit to function. *Coordination* is the act of integrating each task with each organizational unit, so the unit contributes to the overall objective. *Control* is the process of adhering to goals, policies, standards, or quality levels. Orchestrating coordination and control need intense and ongoing communication. Communication is the exchange of complete and unambiguous information—that is, the sender and receiver can reach a common understanding. Distance exacerbates coordination and control problems directly or, indirectly through its negative effects on communication (Carmel & Agarwal, 2001).
Teams involved in distributed development, heavily rely on information and communication technologies (ICT) for communication and collaboration (Thomas & Bostrom, 2008). Teams can use web-conferencing, video-conferencing, and web-cams along with application sharing software to reduce the communication overhead. Along with these tools, asynchronous modes of communication like e-mail or work item tracking tools are also used to facilitate communication when teams are located in non-overlapping time-zone locations (Miller, 2008).

Along with the benefits obtained through DSD (or GSD), there are many difficulties faced by various organizations. These problems are caused mainly due to distance, time zone difference, cultural differences and lack of trust (Carmel, 1999; Krishna, Sahay & Walshman, 2004). Distributed development faces various challenges like communication, collaboration and socio-culture distance which may impact the project (Hossain et al., 2009). Engineers, managers, and executives face formidable challenges on many levels, from technical to social and cultural (Herbsleb et al., 2001).

2.4 Distributed Agile Development (DAD)

As discussed in the sections above, IT organizations have to continue to enhance their processes to better respond to changing business needs, reduce costs, improve quality and improve time-to-market. In order to achieve such objectives, some organizations are blending the distributed software development with agile approach. Although there are benefits of combining the two approaches, but still executing distributed agile projects is a complex undertaking, particularly on large projects (Thoughtworks, 2008).

2.4.1 Benefits of Distributed Agile Development (DAD)

Agile methods can be beneficial when combined with distributed development. There are studies which show that agile principles help in overcoming some challenges faced by distributed development (Fowler, 2006; Sutherland, Viktorov, Blount & Puntikov, 2007; Kircher, Jain, Corsaro & Levin, 2001; Sureshchandra & Shrinivasavadhani, 2008). The advantages of agile GSD are as follows:
• Distributed software development (or DSD) seems to cause decreased visibility of project status and agile process based on short continuous iterations make it easier to see the problems in the early stages of the project (Smits, 2007; Paasivaara & Lassenius, 2006).
• Continuous integration of software code, which is a central part of agile methods, also helps to reduce software configuration management issues (Lehtonen, 2009).
• Use of agile principles seems to have a positive effect on communication between teams as development in cycles makes it easier for participants to see the short term goals (Pikkarainen, Haikara, Salo, Abrahamson, & Still, 2008).
• Sprint reviews can be an effective way to improve communication as it helps the stakeholders to share information about the feature and requirement dependencies. Although agile methods are a big challenge to communication in DSD, they can also be a major benefactor as they emphasize on communication and provide useful communication practices (Paasivaara & Lassenius, 2006).
• Agile principles can even help in creating trust between different cultures involved in the process by constant communication and delivery of software (Paasivaara & Lassenius, 2006).
• Due to increased communication and collaboration there is an improvement in the software quality and increase in the team motivation (Paasivaara, Durasiewicz & Lassenius, 2008).

Thus, using agile in distributed teams has proved to be beneficial for project’s quality and performance.

2.4.2 Risks in Distributed Agile Development (DAD)
Although, handling distributed projects itself is not a straightforward task, combining agile with it makes the process even more complicated (Jalali & Wohlin, 2010). DSD and agile work on different principles which makes the distributed agile projects difficult to manage. DSD requires formal communication amongst the teams which are distributed across the globe. Agile, on the other hand is based on informal communication with co-located teams working in close collaboration. One of the study on distributed agile has summarized the potential conflicts between DSD and Agile, such as communication need vs. communication impedance, fixed vs.
evolving quality requirements, people vs. process oriented control, formal vs. informal agreements, and lack of team cohesion.

Distributed projects face the challenge of spatial, temporal and goal distribution due to large distance, time zone differences (pre-dominantly in case of GSD) and difference in each development centre’s goals (Persson, Mathiassen, Boeg, & Madsen, 2009). Spatial distribution weakens the social relations, limits face-to-face interactions and makes it difficult for the project manager to track individual contribution to the project progress. Temporal distribution makes coordination difficult, causes unproductive delays due to time setting problems. Goal distribution occurs due to faulty information transfer or more concentration on own site's work. It leads to conflicts related to task interpretation, process principles and problem resolution approaches and ultimately low performance and inter site wars (Persson et al., 2009). Other DSD factors like large number of development centers, lack of tool support and infrastructure, dependencies between distributed teams, lack of knowledge base to store the learning’s also impact the development heavily (Hossain et al., 2009). Besides this, geographically separated teams also suffer due lack of effective communication on account of socio-culture difference (Herbsleb et al., 2001).

Moreover, several agile best practices (including collaboration, face to face communication, self organizing teams, retrospectives, showcases, etc) become more challenging in the distributed model (Thoughtworks, 2008). These challenges may impact the project communication, coordination and collaboration processes, hence posing significant risks which needs to be considered for successful completion of the project. Hence, using DSD in an agile environment makes the software development process risky and difficult to manage. Thus, risk management becomes a critical project management activity, which can proactively help in preventing problems in a continuous and concurrent manner (Prikladnicki & Yamaguti, 2004).

2.5 Risk Management Concepts
Since, the objective of this study is to identify risks in distributed agile development (DAD) projects, it is necessary to discuss the theoretical concepts related to risk management.
As per the Webster’s dictionary risk is defined as “the possibility of loss or injury”. Risk is defined as the net impact of the exercise of vulnerability, considering both the probability and the impact of occurrence (Boehm, 1991). Since, we are studying risk involved in software development; risk can be defined in the context of projects. Project risks can be defined as uncertain events or conditions that, if occur, have a positive or negative effect on at least one of the project objectives such as time, cost, scope or quality (Project Management Institute, Inc, 2004). A risk factor can be defined as a condition that can present a serious threat to the successful completion of a software development project (Schmidt, Lyytinen, Keil & Cule, 2001). Risk management is the process of identifying risk, assessing risk and taking steps to reduce the risks to an acceptable level (Stoneburner, Goguen & Feringa, 2002).

A risk can be assessed by using the following terms: Risk exposure: \( RE = P(O) \times L(UO) \), where \( P(O) \) refers to the probability of unsatisfactory outcome, and \( L(UO) \) is the loss to the parties affected if the outcome is unsatisfactory. For a particular project, several classes of participants (customer, developer, user, and maintainer) are involved, each with their own perspective of ‘unsatisfactory outcome’ (Boehm, 1991).

Since risks are present in developments of all types of software, it makes sense to manage it systematically through additional steps in the software development process (Charette, 1989). Risk management involves two main steps (i.e. risk assessment and risk control), each with further three subsidiary steps

1. Risk assessment:
   a. Risk identification: This involves creation of a list of project –specific risk items likely to impact the project success. Various risk identification techniques used are interviewing, decomposition, assumption analysis, risk taxonomies.
   b. Risk analysis: In this phase, the probability of risk and the loss magnitude for each identified risk item are determined to obtain the risk exposure. Techniques involved for risk analysis include performance models, network analysis, statistical decision tree and quality factor (like reliability and security) analysis (Boehm, 1991).
   c. Risk prioritization: Based on the results of the risk analysis phase, a list of prioritized risks is obtained. The prioritized list is used to identify the list of risks, which need
additional planning and action. Less important risks can be documented and tracked for future (Westfall, n.d).

2. Risk control
   a. Risk management planning: Prepares the project managers and team members to address each risk item (Boehm, 1991). Here, one or more strategies for managing each risk are selected, and actions plans are created to carry them out. Basic strategies are risk mitigation, risk avoidance, risk transfer and risk acceptance. The selected strategies must be feasible, appropriate, and acceptable (Charette, 1989).
   b. Risk resolution: This produces a situation in which the risk items are eliminated or otherwise resolved (e.g. risk avoidance by relaxation of requirements). Popular techniques used are prototypes, simulations, benchmarks, incremental development (Boehm, 1991).
   c. Risk monitoring: Involves keeping a track of the impact of risk-reduction implementation. It involves techniques like milestone tracking, follow-up of the top-10 risks and re-assessment of the risk items or corrective actions (Boehm, 1991). Various metrics like Gantt chart, earned value measure and budget and resource metrics

These are the risk management techniques which are applicable to any software development project and hence have been used as the basis of our research work in DAD projects also.

Software risk management approaches focus on ambiguous losses (Boehm, 1991), they rely on multidimensional models that make the management task simple and predictable. They seek to avoid risks, exclude bad alternatives rather optimizing the expected returns (Boehm, 1991). Risk management strategies use observations from the past, they learn from analogical situations, and they use deductive reasoning (March, Sproull & Tamuz, 1991) to detect risky events. The risky items are combined with risk identification and analysis techniques to scan and make sense of the environments. These techniques help the managers to formulate schematic plans for managerial interventional and hence decrease the likely impact of risk incidents, or avoid it altogether.
2.6 Theoretical Model for Risk Categorization: Leavitt’s Model of Organizational Change

There are many studies which apply organizational theories to software development (Beath 1983, 1987). These studies draw primarily on theories which consider task uncertainty, complexity as important organizational design parameters and apply them to software development organizations (Galbraith, 1977). But these models do not consider the sources and forms of risks, nor do they suggest interventions for handling the risks. Hence, a richer organizational model of software development and software risk i.e. Leavitt’s model, is being widely used in IS research.

As a part of this study, we have tried to provide a comprehensive view of various risks associated with distributed agile software development along with various dimensions like causes, origin and methods to control the risks. A need was felt to devise a mechanism for the assimilation of identified risks in organizational context. This will enhance the relevance of the study to the practitioners and the researchers in terms of the specific aspects that need focus and eventually, would reduce the complexity caused by the vastness of the influences in DAD domain. Leavitt’s model of organizational change was found to be a suitable basic theoretical framework for categorizing the identified risks in DAD projects in this study. It has gained significant acceptance in organization theory (Scott, 1992) as well as in information systems (Bostrom & Heinen, 1977; Keen, 1981).

According to Leavitt’s model, an organization is composed of four interacting components, namely, task, structure, actor and technology (Leavitt, 1964), as shown in Figure 2.3. These components can easily be translated into well known elements of software development. ‘Task’ signifies expected outcomes in terms of goals and deliverables. ‘Actors’ cover all stakeholders including users, managers and designers. ‘Structure’ denotes project organization and other institutional arrangements. ‘Technology’ means development tools, methods and hardware and software platforms. The key conjecture of this model is that, the components are strongly interrelated: change in one component will have effects on the others. This mutual dependency is depicted by the edges that connect every component to the other, hence creating the diamond shape. Further, if one component is in contrast with the other component, it may lead to
considerable dysfunctions and hence reduced performance of the system. If this unbalance persists for a longer period of time, it will ultimately reduce the capability to perform the task, and eventually undermine the existence of the whole system.

Leavitt’s model was chosen due to its intuitive appeal, simplicity and its appropriateness for the problem situation. The risks which will be identified through the exploration were expected to fall in categories that align with the major dimensions of the Leavitt’s model easily. Other than alignment of risk categories with the components of Leavitt’s model, significant reasons for using this model as the base theoretical framework for rationalizing the findings of the exploration were (Lyytinen et al., 1998):

- Software development organizations share common characteristics, structuring principles, environmental adaptation and scanning techniques, leadership patterns, and change dynamics as any other business organization. Therefore, Leavitt’s model is equally applicable to software development organizations.
- Leavitt’s model is sufficiently broad as to take into account the most essential aspects in software development.
- Leavitt’s model has been extensively used in Information Systems literature and hence, is widely known.

Despite its merits in depicting organizational components effectively, Leavitt’s Model has been criticized for not considering certain other organizational dimensions. According to Grant & Mergen (1996), although, Leavitt’s Model addresses organizational communication under ‘structure’ component but does not consider the important aspects like technical communication (between the technologies) and social communication (between the people). They modified Leavitt’s model by adding one more dimension, i.e., ‘communication’. Similarly, Kwon and Zmud (1987) have augmented Leavitt’s model with the concept of environment and Davis et al. (1992) included culture as a dimension for making it more inclusive.

Although, this model also possesses certain weaknesses, but due to its suitability to the findings of our study, we preferred it over other models.
2.7 Consolidation of Literature Review

In this chapter we reviewed the benefits and issues concerned with distributed agile development. The review of existing studies reveals that distributed agile development has gained attention due to its potential benefits such as shorter time to market, reduced development cost, and managing late requirements’ change. Although several organizations have reported a success in combining DSD with agile practices, there are significant challenges too. The major difficulties are summarized as related to communication, personnel, culture, different time zones, trust, and knowledge management. These problems predominantly arise due to the conflict in the basic principles of distributed development and agile development. Distributed projects are primarily characterized by spatial distance between the stakeholders, temporal distance, difference in the work/development culture, differences in languages used and large size of the projects. Major values on which agile approaches rely are, close collaboration between the customer, frequent face-to-face communication within the team members, delivering working software within time and budget constraints instead of comprehensive documentation. Distributed development depends on formal communication and detailed planning which does not align with the informal approach used by agile methods, hence posing a real challenge to DAD projects. We did an extensive literature survey to find out the challenges faced by teams executing DAD projects and the suggested approaches to manage those problems. Along with this we also reviewed the literature which considered the issues in distributed development, agile development and traditional software development.
We found that there is scarcity of research work being done in this area and there are certain issues which have not yet been explored at all. The following studies considered for review:

1. Systematic review of literature on Distributed Agile Development: (Jalali & Wohlin 2010), (Hossain et al., 2009).

2. Case Studies in Distributed Agile Development: (Simon, 2002), (Fowler, 2006), (Mattsson et al., 2010), (Miller, 2008), (Lee, Banerjee, Lim, Kumar, Hillegersberg, & Wei, 2006), (Ramesh et al., 2006), (Therrien, 2008), (Dyba & Dingsøyr, 2008).

3. Risk Management Frameworks or Risk Classification for Distributed Agile Development: (Hossain et al., 2009), (Mudumba & Lee, 2010), (Mattsson et al., 2010).

4. Risk Management studies in Agile Development: (Nyfjord & Mattsson, 2008), (Boehm & Turner, 2005), (Misra, Kumar, & Kumar, 2009), (Chow & Cao, 2008).

5. Risks Management studies in Distributed Software Development: (Prikladnicki et al., 2006), (Prikladniki et al., 2003), (Prikladnicki & Yamaguti, 2004), (Persson, 2009), (Jiménez, Piattini & Vizcaíno, 2009), (Carmel & Agarwal, 2001), (Herbsleb, 2007), (Šmite, Wohlin, Gorschek & Feldt, 2009), (Karolak, 1998).


Major limitations of the existing studies are as follows:

1. The study of the existing work clearly shows that there is scarcity of research work which gives a comprehensive view of the risks in distributed agile development environment.

2. There are few systematic reviews studies, which consolidate the risks faced by teams executing DAD projects from the existing research. Although they provide significant challenges in DAD environment, and also discuss about the risk management approaches
for controlling them, but are not able to provide the current issues faced by the industry from the practitioner’s views.

3. There are several case studies or papers which are consolidating multiple cases for DAD projects, but they provide only situation based specific aspects of risks in DAD project. Hence, although these studies bring the practitioner’ perspective of risk management in DAD, but they are not comprehensive.

4. These existing studies on DAD risk management are concentrating on the certain limited number of risks especially those related to lack of effective communication, reduced collaboration and trust amongst the customer and developer. There are other aspects of risks in software development in DAD, for example, software development activities risks, project management related issues, problems related to interaction between the teams and external entities like vendors and third parties, risks occurring due to lack of infrastructure & technology or unavailability of skill to use that technology etc. Although certain studies address these issues with respect to DAD projects, but these studies are scattered which makes these issues less visible or apparent.

5. There are studies which address the risks in distributed development or agile development individually, but very few studies have taken up the issues and challenges which occur when we combine these approaches in significant depth.

Hence, this study has been undertaken in order to unveil the most important risks and challenges faced by distributed agile teams and suggest the appropriate risk management methods to reduce the impact of those risks. We address five major aspects of software development in DAD namely, Software Development Life Cycle, Project Management, Group Awareness, External Stakeholder Collaboration and Technology Setup. Further, we specifically analyze the risk factors occurring due to the use of distributed agile approach by identifying the properties of distributed development which are contradicting with the agile principles. DSD properties which become the root cause of risks arising in DAD projects are spatial distance, temporal distance, work/development culture, language difference and large scope of the projects undertaken.