Creating and Sustaining Software Knowledge
Abstract

Software processes are complex combination of technology and skill, highly dependent on human knowledge. Human involvement and the very nature of software development processes incur high degree of volatility in software processes. Valuable knowledge is created and also lost as software organizations lean towards ‘agility’. Capturing and sustaining this knowledge in readily available and usable form is very important in ensuring continued success for the organization. Software organizations relying on agile methods, usually overlook the importance of knowledge sustenance, which can lead into loss of valuable software knowledge.

This thesis outlines the factors influencing knowledge transfer in today’s increasingly agile software world by carrying out participatory (active) observations in a product based software organization. Knowledge representation forms (text / visual), software architecture, development practices and standards compliance affect how knowledge is sustained within a team and hence dictates the efficiency of transfer to new members.

Keywords
Software knowledge, Software Process knowledge, organizational knowledge, knowledge transfer, knowledge management, software knowledge sharing. Tacit software knowledge, Agile knowledge, Visual knowledge representation.
Foreword

This thesis has been written as part of my GS3D program in University of Oulu. I would like to present my gratitude to Mr. Ben Li, who as my supervisor, supported and guided my thesis work with utmost friendliness. Never at any point did I feel stressed under his supervision. Also I would like to thank my team mates who gave me ideas and at times informed of incidents of interests to my thesis.

I hope to have created some insights into the subject of knowledge sustenance in agile software development through this thesis and look forward to someone if not myself to farther the research in this area.

Syed Mansoor Ali.

Oulu, April 1, 2014
Contents
Abstract .................................................................................................................. 2
Foreword .................................................................................................................. 3
Introduction ............................................................................................................. 5
1.1 Knowledge and Software Organizations ...................................................... 6
1.2 Research Problem / Objective .................................................................... 6
1.3 Research Method .......................................................................................... 9
    Participatory Research & Field research ......................................................... 10
2. Background ........................................................................................................ 11
2.1 What is Software Knowledge? ................................................................. 12
2.2 Software Knowledge Management .......................................................... 14
2.3 Software Conformance and Consistency .................................................. 15
2.4 Knowledge Obstacles ................................................................................. 16
    Non-Compliance with Standards .................................................................. 17
    Technological Debt ...................................................................................... 18
2.5 Knowledge Visibility .................................................................................. 18
2.6 Knowledge Capture and Dissemination ..................................................... 20
    Software Visualizations .............................................................................. 20
2.7 Utility of Visualizations in Software Knowledge Transfer ....................... 21
2.8 Visibility vs Visualizations ......................................................................... 24
2.9 Improving Software Process Visibility / Awareness ................................. 24
    Code map visualizer ...................................................................................... 25
    Experimental Settings for Software Knowledge Sharing ............................ 26
3. Research Method .............................................................................................. 27
3.1 Participatory / Field Research .................................................................... 27
    Types of Participatory Research ................................................................... 29
    Limitations of Participatory Research .......................................................... 30
4. About Product-A ............................................................................................... 31
4.1 Early Development ..................................................................................... 31
4.2 Second Phase of Development ................................................................... 32
4.3 Development Process ............................................................................... 32
4.4 Knowledge Transfer to New Team Members ........................................... 33
5. Utility of knowledge management .................................................................... 35
5.1 Observations ............................................................................................... 35
6. Discussion ......................................................................................................... 47
6.1 Agility and Knowledge Creation ............................................................... 48
6.2 Effects of Distributed Team Structure & Agility ....................................... 51
6.3 Good Design & Learning .......................................................................... 54
6.4 Technical Illustrations vs Verbose Descriptions ....................................... 56
6.5 Team Mavericks & Technical Capacity ....................................................... 58
7. Contributions to Agility ................................................................................... 61
8. Limitations of Study & Further Research ..................................................... 65
9. Conclusion ........................................................................................................ 67
References ............................................................................................................. 70
Appendix ................................................................................................................ 76
Introduction

Software industry has been trying to streamline development processes for at least the last two decades; slow but steady progress is being made. Yet a look at historical performance of software development processes reveals that improvements are still far from efficient; for instance figures provided by literature (Laird & Brennan, 2006):

1. 23% of all software projects are cancelled before completion.

2. Among projects that are completed only 28% were on time, within time and with all originally specified features.

3. Average software project overran budget by 45%.

These rather disappointing figures are largely a result of bad management of development processes. In practical software development, it is very easy for a process to slip into delays, non-conformance with the original plan, non-conformance with the required functionality or incapacity to scale the software with the functional evolution. Especially in software businesses where code footprint becomes medium to large sized. Any software system spanning multiple development units (software projects), thousands of lines of code and/or comprised of distributed software artefacts can be considered medium/large sized. Other attributes of medium/large software systems include number of features and functionalities it supports.

Another factor affecting efficiency starts surfacing: managing the software knowledge. Why is managing software knowledge that difficult? To answer that question one needs only to look at the nature of the software development; especially when the software being developed is large, distributed or being developed by distributed teams. Knowledge created as a result merits proper attention for sustenance. The information that needs to be reviewed in order to make educated decisions grows; eventually the scale of information becomes too large for effective and timely reviews. This information overload has negative effects on human cognition thus causing slippages in performance.

In software organizations one of the most valuable assets if not the only one, is knowledge. This knowledge is usually synonymous with individuals who create, absorb and possess the knowledge. For smooth operations and growth of a software organization this knowledge must be properly retained. Software development is essentially a knowledge-based industry. When technological change or radical disruption impacts on organizations or industries, which haven’t invested in associated technological fields, they often fall into problems, because they might not be able to assimilate new knowledge and would have to transform their knowledge structures. Getting locked into old technologies can be a result of not investing in acquisition of new external knowledge. (Cohen & Levinthal, 1990)

From the perspective of software development organizations and my subject of interest, the above observation by Cohen and Levinthal refers not only to new knowledge in terms of new technology but can also be seen to point to the knowledge that exists within the organization in isolated form. Software development depends on highly
individual skill and this knowledge can easily get compartmentalized/isolated, which makes knowledge preservation a very difficult task for the organization. In my research work I have used observatory / field data to analyze the cause and effect relationships between organizational practices and knowledge transfer in software organizations.

In organizations where knowledge holds such value, employee turnover can potentially leave visible effects on efficiency. For instance, a software module developed by an individual, may leave very little documented knowledge for others in his permanent absence. To ensure continued efficiency, software organizations must bring new employees up to speed as quickly as possible i.e. transfer knowledge to them. Software knowledge creation and transfer is the focal point of interest of this thesis. Although both of these concepts are influenced by socio-technical and cultural aspects of an organization this thesis focuses on influences cast by software processes/practices.

1.1 Knowledge and Software Organizations

Knowledge is defined as justified truth. (Nonaka & Takeuchi, 1995), a belief or collection of beliefs that are widely understood and attributed as valid and have practical or theoretical utility value. Nonaka and Takeuchi in their book ‘The Knowledge Creating Company’ give an operational definition of knowledge, as the capability of a company as a whole to create new knowledge, disseminate it throughout the organization, and embody it in products, services and systems. (Nonaka & Takeuchi, 1995)

Given that the nature of software development it is widely recognized as a knowledge intensive profession. The software engineering field is multidisciplinary and inclined towards people as the retainers and appliers of knowledge as software artefacts (Boehm, 2006). Unlike manufacturing industry or assembly lines, software production requires much more human intervention on individual and collective levels. Developing software of any scale is highly dependent on knowledge. Two major classes of this knowledge can be seen:

- Technical knowledge about developing software (design, architecture, problem-solution)
- Application knowledge about the purpose of software (domain/business knowledge)

Tacit and explicit knowledge are two terms found in literature which separate and define (Nonaka, 1994). Both categories listed above can be sub-classified into tacit and explicit knowledge; where tacit knowledge refers to intangible knowledge (i.e. knowledge about software held by individuals with or without their awareness of it) and explicit knowledge refers to knowledge that exists in tangible forms like design documents or other technical material.

1.2 Research Problem / Objective

Medium to large scale software products are particularly at risk of running into a phase where it can partially or wholly develop blind corners within i.e. parts of software that
no one understands or has knowledge about. Such cases can eventually incur unnecessary cost for the business and sometimes even have fatal effects. When referring to ‘knowledge’ in context of software products, I see it in following categories based on extant literature (Ko, DeLine, & Venolia, 2007). These categories are technical segmentation of the broader term of software knowledge.

- Business functions knowledge
- Architectural knowledge
- Implementation knowledge
- Problem-solution knowledge

Business functions knowledge is the understanding of what utility purposes a software fulfils. For instance a financial calculations software’s business functions can typically include tax calculator. Architectural knowledge refers to the knowledge of high level structure of a software system, platforms utilized, external and internal dependencies and other constituent components.

Implementation knowledge is the knowledge segment that defines and describes the source code level understanding of a software system which includes the exact knowledge of structure and organization of source code. The problem-solution knowledge is concept is more of a sub-category for implementation knowledge, this is the understanding of why a problem occurred and how was it solved and how can it be avoided in future.

All of the areas listed above are composed of two classes of knowledge as mentioned earlier, Tacit and Explicit knowledge. (Ichijo & Nonaka, 2006; Nonaka, 1994). Sustaining tacit knowledge can be very challenging, is it does not have any tangible state nor can one tell what knowledge exists in tacit form. Tacit knowledge involves human processes in knowledge management - creativity, conversation, judgment, teaching, and learning - and it is difficult to quantify; therefore, it is difficult to manage in the traditional disciplines, which are more quantitative than qualitative (Ichijo & Nonaka, 2006).

Software companies often run into risks of losing tacit knowledge with employee turnover or team re-structuring. Because tacit knowledge cannot be managed like tangible resources it becomes one major source of most problems software organizations face i.e. in software systems development, because of high knowledge dependence. Explicit knowledge on the other hand is relatively easily maintained because of its tangible form of access i.e. documents, emails, messaging logs or audio or video records. Sustaining explicit knowledge is usually a routine task of archiving skills.

How should the software organization transfer its existing knowledge to this new person? Knowledge that has never been replicated except in the minds of individuals who have been there before, knowledge that has been recorded but is present in an form that pose comprehensibility challenges or is so hard to access that it loses its efficiency. These and other similar questions surround new team members: what to know, where to know it from? Who can tell me?
More often than not software organizations find themselves beset with problems of delayed and cancelled projects, inadequate quality, long cycle times and high costs. These problems are compounded by market pressure, competitors and the need of getting new products/versions out (Ravichandran & Rai, 2003); Ravichandran & Rai’s list of problem causing factors do not account for technological debt and employee turnover; however technological debt and employee turnover do influence knowledge transfer efficacy as will be discussed later on. Software organizations to an extent counter the above problems with different software development methodologies.

Moreover the ‘completeness’ of information is often questionable. In addition to knowledge of programming platforms and languages, one requires current and historical information about the software, evolution of a particular piece of software over a certain period of time and problem-solution relationships as well. This knowledge becomes a valuable asset for any software organization, retaining and transferring this knowledge dictates the success or at least the efficiency of software organizations.

Factors like uncontrolled software growth or employee turnover can seriously harm quality and efficiency of organizational knowledge. As the software projects advance, the scale of information they create can become taxing on efficiency; therefore requiring an appropriate process for sustaining and transferring this knowledge. Software development produces both tacit and explicit knowledge, both types of knowledge is significant but it is the tacit knowledge that can become very expensive liability for a software organization.

Hence the need for a means to easily and efficiently distribute large amounts of information and more importantly make sense out of it. The most inefficient method of this information sharing is the source code itself, void of any supporting material, source code can become a nightmare of a product. Software knowledge can be lost or become very difficult to salvage.

Through this thesis I have attempted to distinguish, highlight and correlate different attributes of a software organization and team that govern the efficiency of knowledge sharing methods in software organizations. Efficiency here refers to how well a knowledge transfer method brings a new team member up to speed, is the information transferred complete to carry out a software task e.g. development of certain feature. In software organizations team members and their work practices can provide insights into knowledge gaps and potential information sharing problems; since team members in all industries play vital role in sharing their knowledge and experiences and are aware of historical events, they can provide valuable information for researchers. M. German and Hindle “It will be valuable to ask developers and other interested parties (such as managers and other researchers) what their needs are and how they believe they can be satisfied” (German & Hindle, 2006). Applying German and Hindle’s thoughts on my research area, I’ve positioned software development team at the core of my observations.

Research Question:

What are the factors influencing efficiency of software knowledge transfer to new members in agile teams?

This thesis work has studied a PSA project (professional services automation), built and operated by a medium sized software company in southern Finnish city of Lappeenranta. I had an opportunity to work full time in the software development team...
which allowed me to closely observe (participatory research) the work process and knowledge management procedures. I will refer to this software product as Product-A, and the company as Company-A for the sake of anonymity and allowing maximum objectivity.

This software product has experienced significant growth in the last few years, starting from handful of features and now containing a multitude of them. Growth in a software product is almost always a synonym for growth of its code base as well and with growing code base knowledge management problems do arise. The most significant part of my study will be about finding an effective way of inducting new people into an existing large codebase product like Product-A. Research objective of this thesis work, is to distil and establish the relationship(s) between forms of software knowledge documentation, team structures, work processes and their influence in knowledge transfer to new team members. Also to examine the cause and effect relationships in a software organization that eventually lead to compartmentalized knowledge and knowledge transfer problems in software businesses by applying literature and observations to Company-A.

**Efficiency:** Efficiency of software knowledge transfer refers to the degree of ease with which a knowledge seeker can comprehend the information presented. Also in context of software development practices, software knowledge and development, efficiency refers to the capacity of a team member as an individual and the team as a whole to absorb and utilize software knowledge in conducting software development tasks (designing, developing, extending and debugging software artefacts).

### 1.3 Research Method

Research conducted for this thesis utilized participatory research method. Participatory observations are made by closely working with a software development team as a team member (software designer); hence achieving high degree of participatory involvement. Working on a full time basis with the software team provided first-hand information and experiences about how knowledge transfer and work coordination was achieved in the company. My data collection method included informal interviews, participatory observations and personal experiences by taking an objective observatory approach on myself as well as a subject who is new to the team. Company-A’s development process was scrum based so daily meetings called daily scrums were held as part of everyday routine, which provided me with observation material. In addition to these meetings, there were also occasional formal meetings held after every release for retrospective; inter-team informal conversations were also a great source of insights. All information gathered during these observations was recorded as notes to myself.

Studying the process of knowledge transfer in a real software development team, allowed to identify factors that determine how well knowledge is transferred to a new team member, how existing documentations, peers and processes support knowledge transfer in Company-A.

The research also contains certain features of field research and case study research methodologies. I have conducted my observation during my employment at Company-A, which allowed me to closely observe and interact with the team working on product-A. I have collected data both as a passive and participant observer.
Participatory Research & Field research

The primary research method I have utilized in this research is participatory research method, where the researcher integrates him/herself into a group (team) to carry out observations and collect data. It is important in this type of research that the subject (team) accepts the researcher as one of them; it is important to become friends, or at least be accepted in the community, in order to obtain quality data (Howell, 1972).

Participatory research requires the researcher to work very closely with individuals comprising part or whole of the research subject. By inducting his or herself into the activities a researcher collects first hand data. According to DeWalt & DeWalt the researcher must strive to fit in with the population of study through moderation of language and participation (DeWalt, DeWalt, & Wayland, 1998). Since I was part of Product-A’s development team, I was able to closely participate, observe for valuable insights and inquire from other team members the data needed. My research also involves data retrieved from informal interviews and casual conversations with the team members and former employees of Company-A. Observational data is recorded in the form of notes to myself.
2. Background

Software development is a knowledge intensive process and hence relies heavily on individuals and their knowledge capacities. This chapter will shed light on the problems software development processes and organizations face in terms of knowledge transfer, especially to new team members. Because of its high dependence on knowledge, software development is also very highly dependent on individuals who act as carriers and keepers of the knowledge. The knowledge dependence in question can create efficiency and quality related problems particularly in context of software knowledge creation, sustenance and transfer. For various reasons individuals may or may not be available to the software organization at all times, either temporarily or permanently which mandates a need for externalizing and maintaining the knowledge. Based on extant literature I will discuss how a software team’s work flow, work quality and work practices can affect the transfer of knowledge during induction of new team members.

The degree of efficiency and quality of software being developed has principal reliance on the quality of existing knowledge within the team and/or organization. Software products usually undergo continuous evolution; new knowledge also gets created as a result. To sustain the quality of software, an organization should consider the evolution aspects of software development teams, because team members are the principal keepers of knowledge that goes into software development. As software evolves the information / knowledge evolves along with it. The combined effect of software and team evolution (or devolution in some cases) creates the need for recording software knowledge.

In absence of proper management of software knowledge, in addition to deterioration of team efficiency and development quality, a project’s review and decision making can also deteriorate; leading to loose management and consequently subpar software development performance. For instance bad software design of a software product can lead to technical lock-ins which will eventually lead to technological debt and knowledge transfer problems. Moreover, a software organization’s context is subject to continuous change, it’s not only the source code that has to be understood for successful operations of the organization, understanding the historical software changes, social context of the software team, problems and details of the solutions also play a vital role. All these factors undergo persistent change as business grows, creating contextual knowledge which should be maintained in an accessible form.

Software and organizational knowledge mentioned above is defined (but not limited to) methods, practices and individuals, which are dependent on time and hence its state does not remain constant. Software ages as this social and organisational context evolves: teams change, knowledge decays, documentation falls out of date, intentions and rationale are forgotten over time (Ball & Eick, 1996). Over the course of time during which the software is being developed, due to the complex nature of development work it becomes difficult to maintain conformance1.

1 requirements conformance, standards conformance, quality conformance and time line conformance
With growth and evolution of a software artefact, very often the issues (software bugs or change requests) related to the software also grow. There can be various reasons behind the occurrences of software bugs, issues or change requests including mismanaged or inaccessible contextual knowledge, inflexible software design which is hard to extend or adapt or non-standard source code implementations combined with team evolution (new members may not perform well due to knowledge access problems). As a result of reasons stated, software growth can potentially become too difficult to handle effectively; as fixing or extending software requires a certain understanding of existing software artefact, if not provided efficiently and effectively the knowledge gaps can greatly hinder a team’s performance in software issue handling and/or extension.

In context of software knowledge and development efficiency and efficacy refer to how quickly an individual can grasp the knowledge required to extend and/or further develop the software artefact in question. In addition to that it also refers to the capacity of the individual to relate separate knowledge assets to create a holistic understanding of the software.

In cases where software source code is the only information that is available, project teams can easily become overwhelmed with cognitive overload and hence end up in a fuzzy work-direction. In order to effectively continue working the issues/bugs have to be continuously handled with high efficacy. One study refers to huge pools of bugs/issues as black hole. (B.Ellis et al., 2007), because they can easily overload the cognitive capacity of a software developer. The same study goes on the mentions responses from more participants referring to bug tracking systems as ‘I feel like I’m constantly drowning!’ which highlights the cognitive overload an individual can face in software development work. This obvious overwhelming and overloading of the cognitive abilities of project members causes adverse project performances, which can potentially lead to severe negative outcomes. There are other contributing factors pertinent to cognitive overload which fall out of scope of the study mentioned i.e. knowledge obstacles. Knowledge obstacles here refer to factors that degrade knowledge absorption processes for software teams. These factors can range from absent documentations to inefficient work processes.

In this thesis I conducted a study of a software organization’s work process and practices and how they influence creation and transfer of software knowledge. The study also includes analysis of other factors like effect of knowledge representational i.e. visual representation of data to reduce cognitive overload and aid human comprehension. Research under discussion here is limited to knowledge related to software development and does not consider other related processes like team management etc. The study also analyzes how distributed teams and agility effect knowledge capture, creation and absorption.

2.1 What is Software Knowledge?

Software development organizations live or die based on how effectively they generate, assimilate, reuse and leverage their knowledge (Desouza, Awazu, & Tiwana, 2006). Like corporate culture and industrial jargons, every software organization inadvertently or otherwise, creates its own knowledge which deeply governs and defines the software product or service that organization produces. Attempts to capture this knowledge in order to transfer it further, falls under management of knowledge. Extant literature
about software knowledge refers to this process of recording knowledge as ‘Codification’ (Ravichandran & Rai, 2003). These knowledge documents provide guidelines and help for new incoming people, allowing them to get accustomed to the software intricacies and contextual knowledge which is very important for efficient software development work.

In software organizations most of the times team members (software development team members) make decisions based on personal knowledge and experience or knowledge gained using informal contacts. This is feasible in small organizations, but as organizations grow and handle a larger volume of information, this process becomes inefficient (Johansson, Hall, & Coquard, 1999). Some authors state that software knowledge can be classified into two high level categories namely Declarative and Procedural knowledge (Im & Hars, 1998; Joereskog & Wold, 1982; Zack, 1999).

Declarative knowledge in the context of software can be defined as knowledge of ‘What’ i.e. the business domain a particular piece of software provides value to, the data, design or concept it encapsulates in the form of a program (Ravichandran & Rai, 2003). Procedural knowledge on the other hand is defined as the knowledge of ‘How’; how does a piece of software utilize certain inputs and transforms them into a value proposing output. Inputs like user requirements, technology and skills get transformed into functional application systems (Ravichandran & Rai, 2003). Declarative knowledge and procedural knowledge can also be referred with more common terms like domain knowledge and technical knowledge respectively and form the key segments of software knowledge necessary for an individual to carry out proper development work.

Im & Hars (Im & Hars, 1998) farther argue that once the externalized knowledge is validated, it can be re-used in subsequent projects, hence sustaining valuable knowledge for a software organization. From a knowledge management perspective knowledge creation and knowledge embedding2 are important antecedents to software process capability (Ravichandran & Rai, 2003).

In software organizations today the knowledge is distributed among stakeholders and there no single point of all knowing software knowledge. Efficiency in software development organizations is achieved by successfully combining and utilizing all facets of knowledge held by all stakeholders. As software systems are getting increasingly complicated today, the knowledge needed for the implementation is vast and unlikely to be held by any individual software developers (Yunwen, Yashuio, & Kouichi, 2004); hence the significance of effectively recording and disseminating software knowledge is larger than ever.

In modern software industry computation is increasingly becoming decentralized (distributed), software is being made to utilize SaaS and SOA philosophies. These factors alone have added a learning curve for newly inducted members in a software team. Ye (Ye, 2006) points out the complexity of today’s software and the degree of knowledge dependence by stating: Even if the software itself might be a simple one, because the computational environment in which it runs consists of large and complicated programs, it still demands substantially more knowledge than applications written in early days (Ye, 2006). Ye includes technological aspects as contributing

---

2 Knowledge embedding is the process of absorbing individually and collectively held knowledge into the organizational processes. (Nonaka, 1994; Zhao, 1998)
factors in knowledge complexity, leaving out the factors related to application of a certain piece of software i.e. business domain, functional logic etc., which are also important attributes of software knowledge in many software organizations.

2.2 Software Knowledge Management

The diversity and disparity in knowledge requirements for software organizations merits proper attention given to its sustenance. Software development is purely a knowledge based process which can only be carried out by skillful knowledgeable individuals. Any organization that depends on smart people and flow of ideas must choose a knowledge management strategy (Hansen, Nohria, & Tierney, 1999). Individuals keep on moving within or from one company to another, but the knowledge they possess must be kept intact within the organization.

From the birth of software industry, software organizations have been expert dependent bodies of work (Blackler, 1995). Hence creating a directly proportional relationship between the performance of the organization and its key member. It is not uncommon for software development units to rely very heavily on a maverick project manager or a highly skilled programmer to successfully complete a software project (Ravichandran & Rai, 2003). Such selective reliance on individuals can lead into knowledge compartmentalization where a part of development team may know not possess appropriate knowledge about the software artefact they work on. Such organizations eventually run into information deadlocks, where knowledge is concentrated in a select few individuals, rendering high risk of valuable knowledge loss, if the individuals leave the organization.

To avoid such deadlocks and compartmentalization software knowledge should be managed as an asset/resource. Knowledge management in software organization is seen as an opportunity to create a common language of understanding among software developers, so they can interact, negotiate and share knowledge and experiences. (Aurum, Jeffery, Wohlin, & Handzic, 2003). Knowledge scales in today’s software paradigms i.e. (SaaS, SOA, distributed computing, cloud computing) is highly distributed and complex and produces large amount of information; finding and utilizing this knowledge at the right time is often overlooked which creates an indifference to availability or non-availability of information. Transferring knowledge experience is more effectively done hands on. (Johansson, Hall, & Coquard, 1999). Research in software engineering has indicated that software development teams are likely to repeat mistakes made earlier or make duplicate efforts to ‘re-invent’ the wheel. (Basili, Lindvall, & Costa, 2001) And (Brössler, 1999).

Written information has also been found to provide little to no value without access to the author (Johansson, Hall, & Coquard, 1999). A study carried out at Microsoft determined that the most difficult to satisfy information needs were about software design questions i.e. the basis of workings of a piece of software including structure and control flow (Ko, DeLine, & Venolia, 2007). They also found that some information seeking was deferred because the co-worker holding the knowledge was unavailable. They do not specify the reason why written knowledge was not referred to in such cases; a simplistic explanation could be tedious nature of reading through text documents or searching information repositories. Ko et.,al.’s work hence shows that in practical work environments knowledge is transferred more through hands on means which allows knowledge seekers to acquire contextual information and experience as
well as direct explanations of his/her query. Experiences are valuable only when they can be contextualized, decontextualized, and re-contextualized at the proper time (Ackerman & Halverson, 1998).

Referring to tacit form of software knowledge in an organization and its importance, Rus and Lindvall (Rus & Lindvall, 2002) state that experienced software developers transfer this knowledge to inexperienced (new) developers on ad-hoc basis through informal meetings i.e. when a task mandates knowledge transfer. This practice creates knowledge gaps because knowledge does not reach everyone in such informal methods. They recognize the significance of informal knowledge transfer practices but also point out the importance of capturing and preserving knowledge to ensure its availability to everyone.

Software knowledge management, can be seen as composed of two major tasks:

- **Externalization of Software knowledge**
- **Software knowledge distribution**

Creation (externalization) of any knowledge dictates the efficacy of its distribution. And since software development is nothing but transformation of knowledge into source code, creation of this knowledge goes a long way in defining and influencing software quality team experiences and performance. Therefore methods utilized to externalize software knowledge and the form chosen to represent it are of utmost importance. Research shows that long verbose material is rarely consulted when it comes to real life work practices (Johansson, Hall, & Coquard, 1999). Apart from this fact, the form of knowledge externalization also depends on which state a software system presently exists, i.e. under development, under extensions or under modifications etc. Complexity of a system should also be considered when choosing form of knowledge externalization. Taking an example of a fairly complex software system, long verbose documents with no diagrams can incur long comprehension times and even then there’s a risk of confusions and misinterpretations.

Representing technical aspects of software in visual form (graphical illustrations) are much more effective in transferring knowledge. Software visualizations can be used to distribute knowledge of concepts and designs with high comprehension rates and lower risks of misinterpretations. A simple definition of software visualizations is “the use of computer-supported, interactive, visual representation of abstract data to amplify cognition” (Card, S and Mackinlay, D., 1999). Visualizations in software engineering field can represent visualizations of algorithms and programs/source code. In short, visualizations are concerned with visualizing the structure, behavior, and evolution of software3 (Diehl, 2007).

### 2.3 Software Conformance and Consistency

Conformance can be a very wide term even in the constraints of software development processes. Literal meaning of conformance is compliance with certain rules, patterns or practices. In context of software design and development Conformance can refer to

---

3 See Appendix
Conformance with requirements, Conformance with original design, Conformance with the timeline, conformance with design standards and Conformance with cost estimates and so on and so forth. The importance of keeping Conformance is paramount in all of these variations listed above.

High degree of conformance can raise the probability of success of a software project. For instance a software project which was originally planned to be implemented using the ‘model view controller’ design pattern should always be tightly bound to its design blueprint; because it usual for a development team to fall into the vicious circle of garbled code. During the course of development process software fixes, developmental changes, team turnover etc. can cause mutations in the original software design. Hence, straying away from the ‘design conformance’. This deviation of conformance can cause wide gaps in knowledge sustenance and transfer unless appropriate measures are taken. Original design changes to something else should be justified with documented proof and similarly every other deviation should also be provided with its documented justification which will not only preserve the decision knowledge for the existing team but also make sure that in case of employee turnover, knowledge will be available for new comers.

Because software is continuously evolving it is very important that the evolution is regulated to preserve the ‘Conformance’ which will also ensure a degree of efficiency. There are many instances in literature that highlight the significant knowledge of evolution of software (Burch, Diehl & Weißgerber, 2005; Collberg, et al. 2003; Gall, Jazayeri and Riva, 1999; Pinzger et al., 2005; Voinea, Telea and van Wijk, 2005). Tracking software evolution is necessary for understanding the impact of accumulated changes on the architecture of a software system, as well as for identifying ‘drift’ in the conceptual design, for detecting problems emergent from the changes, and for detecting historical trends in software evolution (Petre & de Quincey, 2006).

### 2.4 Knowledge Obstacles

As mentioned in previous section, software organizations very often rely on a select few or sometimes even a single individual for its software development; which creates knowledge sustenance problems i.e. recording software knowledge and then distributing it to other team members, especially to members who are new. Anderson et al. identify such circumstances create inherently unreliable work system where consistent attainment of software development goals cannot be guaranteed. (Anderson, Rungtusanthanam, & Shcroeder, 1994)

They further go on to mention a remedy, by purposefully encouraging knowledge creation and utilization throughout the organizations as opposed to having knowledge and knowledge creation to be in the domain of experts who then selectively disseminate and apply knowledge as needed. Compartmentalized knowledge is a time bomb that guarantees to push a software development team into turmoil. Isolating knowledge in any form puts the organization at risk of losing its benefits (Hansen, Nohria, & Tierney, 1999).

Knowledge compartmentalization is the culmination of a number of factors which go long into the history of a software organization, for instance the failure to record and distribute knowledge right from the start, failure to create multiple replaceable human resources etc.; some authors also mention social problems, when it comes to knowledge sharing in software organizations
Software developers may not know to whom they can turn to for help on this particular problem. (Becks, Reichling, & Wulf, 2004)

Experts who are able to help may not be willing to, due to the interruption to their own work and other various reasons (Cross & Borgatti, 2004)

The effect of facts listed above becomes stronger when the subject is a new team member; being a new member s/he does not have the same level of familiarity with other members as older team members do, secondly if experts are unwilling to help old team members they might be coaxed into doing so somehow since older team members are likely to be aware of personal traits of this expert. On the other hand, a new team member cannot expect or at least has very low probability of ‘out of the way’ favors especially from ‘expert’ team members.

To record and share knowledge some companies rely on tools like wiki or a knowledge base where all recorded information is stored, while the real value of information/knowledge sharing comes from connecting people, who can share their knowledge on the spot. (Stewart, 1998). Johansson et al., (Johansson et al., 1999) note that experiences are best transferred through face to face communications.

Non-Compliance with Standards

A negatively contributing factor in effective knowledge transfer is non-compliance with standards, as it damages ‘inference’ capacity of an individual. Standards here can refer to standard practices that are followed in the software industry. For instance a MVC styled software system should strictly follow the design cues of MVC, hence allowing a certain degree of ‘anticipated awareness’ to the individual exploring and understanding a software system. Software industry exclusively relies on human capital because of the nature of work required. It has not yet reached the automated nature of manufacturing industry yet. Designing and developing software heavily relies on individuals and their knowledge, skills and vision. In past few decades software industry has seen emergence of design patterns as an example of knowledge accumulation within the industry for common problem-solution scenarios. Apart from the patterns and practices followed in this industry each technological platform itself follows or at least recommends certain standards for developing software solutions. These standards, practices and patterns are all form the unseen and untouched phenomenon of software industry’s standard practices.

For further clarification of what exactly an industry standard or recommendation refers to, one can look at the world wide web consortiums documents regarding web standards, CSS standards:

http://www.w3.org/standards/

As well as vendor specific ones i.e. Microsoft’s web .NET platforms design guidelines


There is no strict binding for software organizations to follow these standards as rules lines set in stone. Yet when we look at reliance of software development on human mind and then the fact that human capital cannot be retained forever, the benefits of standards compliance become clearer i.e. knowledge created in standards compliant
software organization will already higher understandability for new team members and help anticipatory exploration (see Appendix).

All new employees in a software organizations, are very likely to be aware of industry standards no matter if they are experienced developers or just starting their career, as educational institutions also prepare their students according to the widespread patterns, practices that form the standards. Therefore when employee turnover brings in people they are more likely to be in sync with technology, development standards and approaches which are widely followed and accepted in the software industry. If an organization is forming its own standards within a software development group and code it is in fact creating a learning curve for new people.

There are exceptions to this notion, but only when a software organization’s scope of influence is wide enough to give birth to new industry-wide standards. Organizations should disregard industry standards only if they themselves are trendsetters and have certain degree of influence over the industry, in any other case doing so can create a time consuming and confusing phase of transition for new individuals being inducted into the development team from outside the organization.

**Technological Debt**

Technological debt refers to an organization’s delayed response to technological evolution and growth which eventually leads to reduced adaptability and sometimes complete incapability to adapt to changing environments. Martin Fowler writes about technical debt: The all too common problem is that development organizations let their debt get out of control and spend most of their future development effort paying crippling interest payments (Fowler, 2014). In software organizations whose core business relies in sales of software both in conventional forms and SaaS paradigm, technological debt can have serious implications. Most significant effects of technological debt can be seen in losing competitive edge to competing products. The capacity of a piece of software to absorb changing technologies and evolve with emerging platforms deteriorates because of these debts and can slowly lead to complete freeze of progress in terms of technological advancement.

In medium to large sized product based software organization technological debt acts more like ‘quick sand’ once a product starts accumulating technological debt, every passing day without a rectifying reaction only increases the costs that will be needed to shed the debt. Technological debt, not only prohibits functional competence of a software but also creates learning limitations for new comers; in the form of either unexpected software architecture, unnecessary dependencies, complex code flow or a complicated solution to a known pattern.

2.5 Knowledge Visibility

Some authors refer to the concept of visibility as ‘awareness’ (Storey, D., & German, 2005; Ko, DeLine, & Venolia, 2007). Dourish and Bellotti (Dourish & Bellotti, 1992) referred to ‘awareness’ as the understanding of the activities of others, which provides a context for one’s own activity; hence providing knowledge to nurture and grow one’s own knowledge. Software visibility means the degree of availability of information and granularity of that information. A software project has high visibility if the stakeholders have access to the status of the process and/or the product starting from the high level
information down to details about every single code module. Conventionally software visibility means investing time and resources into producing documentations that convey related information. The process of creating these documents took time of course, but provided great value for sustaining knowledge for future use.

With the advent of agile methods which do not actively advocate documentations, many organizations completely disregard creating and keeping any software documents at all; hence creating a gaping hole in knowledge sustenance process. While on the other hand also disrupting knowledge visibility seriously.

To add a further dimension of hindrance to visibility think of geographically dispersed software teams and absence of documentations. When a software designer is not required to produce any technical documents to describe the design approach he has taken, the Problem-solution description that he has solved, there is literally no way to record his knowledge for use in future.

There are different solutions that organizations come up with to solve this knowledge visibility problem, HP’s (Hewlett-Packard) for instance, encourages person to person meetings for sharing knowledge and creating new knowledge (Hansen, Nohria, & Tierney, 1999), this may seems like counter intuitive to cost savings but apparently creates enough value for HP.

With agile methodologies’ minimal or non-existent documentation the process and software itself are the only points of reference software teams. Distributed teams have the added problem that they cannot always have in person meetings with all team members and hence strictly rely on the source code for knowledge extraction. Improving the knowledge (contextual and related knowledge of a source code) visibility should provide contextual as well as target specific information.

Prior software development methodologies with more importance given to information creation (regarding the process and software i.e. code documentation) were high in visibility, as illustrated in the following table:

<table>
<thead>
<tr>
<th>Process Model</th>
<th>Process Visibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Waterfall Model</td>
<td>Good visibility, each activity produces some deliverable</td>
</tr>
<tr>
<td>Evolutionary Model</td>
<td>Poor visibility, uneconomic to produce documents during rapid iteration</td>
</tr>
<tr>
<td>Formal Transformations</td>
<td>Good visibility, documents must be produced from each phase for process to continue</td>
</tr>
<tr>
<td>Reuse-oriented Development</td>
<td>Moderate visibility, it may be artificial to produce documents describing reuse and reusable components</td>
</tr>
<tr>
<td>Spiral Model</td>
<td>Good visibility, each segment and each ring of the spiral should produce some documents</td>
</tr>
</tbody>
</table>
However process visibility may be inversely related to rapidity (Sommerville, 1995). Rapidity is synonymous with agility here; therefore pointing to the paradox of agile methods and visibility. In an attempt to attain agility software organizations may lose knowledge visibility and appropriate sustenance of knowledge; loss of knowledge visibility means increased risk of bad decisions, software quality deterioration or slippage in performance and perhaps most importantly the loss of knowledge sustenance means.

2.6 Knowledge Capture and Dissemination

When one compares visual knowledge representation with textual knowledge there is scientific evidence that visual depiction of knowledge is much more effective than textual representations (Larkin & Simon, 1987; German & Hindle, 2006). Especially in the software industry where a small flowchart can tell a whole story which may require hundreds of lines of text. Also elsewhere in the knowledge transfer businesses like publishing the potential of visual knowledge representation has been noted; Based on the efficiency of graphical information representation a whole book series has been launched called ‘HEAD FIRST’. This series of books relies on the idea that knowledge dispersal attempted with text is less likely to be captured by its audience compared to the visual graphical representations.

Software Visualizations

According to Gershon and Page, information visualization is a process that transforms data, information and knowledge into a form that relies on the human visual system to perceive its embedded information. Software visualizations serve the exact same purpose (Gershon & Page, 2001). There are a number of different attributes of software that can be visualized static, dynamic and evolutionary.

The most basic purpose of software visualizations is to convey information in an easy, instantly understandable and least ambiguous way. Its goal is to enable the user/viewer to observe, understand and make sense of the information. (Gershon & Page, 2001). Why rely on visualizations for that purpose when the root information is already available in textual/numeric form? The answer to this question lies in the simple phenomenon of why is it easier for us humans to remember mnemonics than numbers or faces compared to their names, how many times it happens that one recognizes a face but struggles to look for an associated name. It must have happened to every one of us at least once that we saw a familiar face but could not remember the name, implying that our memory storage could recall the photographic memory but not the attributes related to it so easily. Same logic can be applied to software visualizations; visual representation of complex data makes it easier to comprehend and more importantly remember. Developers have to understand the underlying design and functionality before being able write any code (Fowler, 2014) and that’s where visual illustrations can assist them.

Software development process, is considered to be a difficult task because of the myriad of perspectives and huge amount of information (depending on the scale of software being developed) that needs to be perceived not only viewed; in order to make better decisions about structuring and programming a software artefact one should gain high degree of understanding of existing software and its context.
Moreover in modern software engineering paradigms which resort to agile methodologies, the notion of minimal documentation is often misunderstood and results in complete absence of design/technical documents. This does not heavily effect collocated software development teams; yet for teams that are geographically distributed or include new members, capturing and understanding existing software knowledge in a timely manner becomes increasingly difficult.

On top of geographical distance the concept of agility also creates knowledge management problems, as in real life the focus of development team shifts towards getting the software out. Therefore software knowledge sharing becomes trickier since in agile organizations where very little emphasis is given to ‘documentation’. Often the code is the only true and accurate description of the functionality of the software’ (Burd & Munro, 1999). Code definitely is the ultimate descriptor of its own functionality but as software systems grow larger and more complex, understanding the code based on just itself is not an easy task, there is much more information that cannot be clearly seen by looking at source code. For example a distributed system (which is the norm in today’s software industry) may have many service end points that are utilized in order to carry out one function, the logic of this one function might as well be distributed. To understand this logic a visual representation of the functionality will be far more comprehensible than digging through source code and manually finding dependencies for further investigation.

Also, information needs for different stakeholders of the software process are established to be different from each other (Jacobs & Marlin, 1995), for instance a software developer will require different information compared to a software architect or a software designer. The visualizations therefore, have to be able to cater to diverse information requirements of different stakeholders and the software industry has relied historically on standards like UML to graphically represent different facets of information.

2.7 Utility of Visualizations in Software Knowledge Transfer

Software design is not an arbitrary task. Successful software design and architecture is only possible when the person doing it possesses deep understanding of technologies, domain and best practices. Business software usually is composed of complex relationships between different blocks of software programs, dependencies which can be external or internal to the code base of a program and methodologies or design patterns. In simple small programs the source code itself describes itself, but when we look at business software in today’s era of distributed software architectures, cloud based solutions and rich internet applications (RIA), the source code is all but straightforward. A software module that enables its users to keep up with the most updated stock prices and keeps them informed about their profit and loss, can be utilizing many external services to gather and present this data to its user.

For a software organization maintaining or providing complex business solutions, it is crucial to keep the design details accessible and understandable in the fastest way possible. More so, when today’s software organizations are leaning towards agility and quick to market approaches and short response times. In real life a software architect is not always immediately accessible to the person who is writing source code, and hence the only available source of information is either the source code itself or peers who might just be at the same level of understanding as the person seeking the information.
Reading through hundreds of lines of code or design manuals hurts the notion of agility on one hand and is taxing on a person's cognitive capacity as well. Is there a better way of representing complex software knowledge?

Prior research on graphical representation of information has shown that it is much more efficient and also does not strain a person's mind. Larkin and Simon have stated that human visual capacity can be exploited to garner better process understanding and interpretation in informational context. (Larkin & Simon, 1987). Visualizations are supposed to rid a person from cognitive overload (German & Hindle, 2006). M. German and Hindle (2006) agree on the capacity of visualizations in reducing information/cognitive overload. Keeping in mind the different information requirements for a software project’s stakeholders, one can clearly see the need for multiple or customizable information visualizations. A user interface developer would require interactive mockups of a system which will help him/her to implement a UI. A back-end developer will require design information about the functioning of a system.

Having a tool that provides multiple visualizations serving all the information requirements can turn into a dilemma i.e. too much information (Storey, D., & German, 2005). The software industry came up with visual representation of software design in the shape of UML. UML is a language/standard that allows a software designer or architect to efficiently and clearly define a software design from multiple perspectives. Each type of UML diagram can highlight a certain facet of a system, hence taking away the need of reading through verbose descriptions.

Creating and maintaining software systems’ design and architectural knowledge in graphical form preserves it and hence makes it potentially visible to the entire development team; provided that the graphical form is widely understood and known like UML or other standards for software design. Since, graphical forms of information are easy to go through, teams are more likely to actually seek information from these documents, as opposed to empirical evidences that state that in real life people rarely look at knowledge bases and documents when they need answers. Software development is highly coordinated activity. Effective coordination requires that information in various forms be exchanged unambiguously among the stakeholders in the development team. Unlike the construction industry, however, there is no useful physical representation of the emerging software system (Wang, 2002). Besides providing quick reference knowledge to active developers these documents will also ensure that the new comers will become productive relatively quickly and easily. (Storey, D., & German, 2005) state that information that is standardized is easier to extract and share, because it works on uniform semantics. The incorporeal nature of software itself inhibits the effective communication of ideas and concepts between individuals (O'Reilly, Bustard, & Morrow, 2005). Standardized graphical representation of software systems, constraint an individual’s interpretation to a fairly narrow field, in contrast to that large description of functionality or software design may still leave considerable room for misinterpretations. Verbose descriptions of a software design can pose the following problems:

- Grammatical room for error, when sentences can hold multiple meanings
- Time to read through all description is long
- Reader’s language skills can also influence his/her understanding.
The problems listed above can easily create a waterfall effect for the software organization, once the person who has written them is not available for other team members, the knowledge which was wrongly perceived can propagate into the team without anyone flagging it. In general, the difficulty of coordinating activities during software development is a major contributor to the costs and failures associated with the development of software systems (Kraut & Streeter, 1995).

Another utility that can be achieved by the use of visualizations is ‘Software Quality’; since software quality is highly dependent on the process used to develop it (Humphrey, 1989); by eradicating ambiguities and reducing the risk of multiple perspectives within a team, graphical representation of software design can improve software quality, improve agility and also contribute valuable information/knowledge assets to the organization.

The utility and efficiency of visual representations of information are attested by Larkin and Simon (Larkin & Simon, 1987). They suggest some benefits of using visual representations compared to textual/numeric information by stating:

- Diagrams can group together all information that is used together. Thus avoiding large amounts of search for the elements needed to make a problem-solving inference.
- Diagrams typically use location to group information about a single element, avoiding the need to match symbolic labels.
- Diagrams automatically support a large number of perceptual inferences, which are extremely easy for humans.

According to him a diagram is better representation because the indexing of this information can support extremely useful and efficient computational processes (Larkin & Simon, 1987).

Software visualizations have the potential to be beneficial in a number of ways. One study for instance carried out an experiment using ‘war room command console’ analogy to erect multiple display screens to mimic a large console identical to command and control centers. Participants (who were all members of software development teams) were asked to hold their stand-up meetings in the ‘command room’. A project manager from this study responds to the utility of this visualization setup as: ‘War Room Command Console’ was useful for bringing new members of a team up to speed with a product’s description, with it been seen as a: ‘major benefit of the system.’ (O'Reilly, Bustard, & Morrow, 2005).
2.8 Visibility vs Visualizations

Visibility and awareness refer to the degree of awareness a team member has about other’s work for example a UI developers knowledge of backend developers work, a team manager’s knowledge of developer’s work etc. Visualizations on the other hand are the tools and means by which information is presented to a knowledge seeker, for instance a code flow diagram, a class diagram or a burn down chart, which all represent different aspects of software knowledge but in visual forms.

2.9 Improving Software Process Visibility / Awareness

Software visualizations are a very well suited means for raising visibility and awareness of a software knowledge because of their intuitive feature of representing information. Information visibility at different levels of organization and stakeholders has historically been addressed through the use of various tools. Many of these tools rely on visualization techniques to either augment existing views in the development environments or to provide specialized views of human activities combined with software artifact information (Storey, D., & German, 2005). Presenting information in a manner that assists clear understanding is the basic benefit; however according to Card, Mackinlay and (Card, Mackinlay, & Shneiderman, 1999) there are six distinct benefits that can be gained from the use of visualizations.

- Increase in memory and processing resources available to the user.
- Reduction of the search for information.
- Enhancement of the detection of patterns.
- Enablement of perceptual inference.
- Use of perceptual attention mechanisms for monitoring.
- The encoding of information in a malleable medium.

The usage benefits of visualization are highly dependent on the context of its use. For instance a project manager’s perspective is naturally going to be different than that of a stock holder or a developer. Never the less, prior research in this area has identified a common benefit as well (Ellis, Wahid, Danis, & Kellogg, 2007) i.e. being able to prioritize work to be done. Once a stakeholder has access to the information provided by the visualization s/he can evaluate and compare each issue based on the current situation of the project and ensure high productivity. One lead put it this way: “First it shows me all unassigned bugs that are in the NEW state. Show me the bugs that we’ve got targeted for the current milestone – making sure the right bugs are targeted and where people are working. Once a month, I look at the ‘show me everything’ query (Ellis, Wahid, Danis, & Kellogg, 2007)

Making decisions about work direction in an evolving software project environment requires much more information than just compartmentalized information of a section of software being developed by a single member. For example just looking at the number of issues related to a project will not bring about any meaningful information until and unless the observer is well aware of the auxiliary information like duration of
the issue, timeline of the project, when is it supposed to be finished, who is assigned to fix the issue under discussion, when is a delivery expected, how many critical issues are present currently and so far and so forth. Even though human brain is very capable of pictorial memory yet providing the information in a visual form extends and amplifies human comprehension of both the immediate and contextual information. Hence making it much easier to make educated decisions.

The ability to tailor the data (via configurable attributes of the visualization gadget) to be visualized allows multi-perspective utility of the visualization. A tool that facilitates multi-perspective understanding is very efficient in work environments which are largely rely on co-ordination. Froehlich and Dourish advocate this notion that such a tool aids coordination not by bringing everyone into alignment with a common perspective, but by providing developers with an enhanced understanding of the work of others and of the group, allowing them to make appropriate decisions about their own activity. (Froehlich & Dourish, 2004)

Based on the extant literature clear superiority of visual representation of software knowledge can now be inferred. To acknowledge this fact and provide the benefits of graphical code representation, one of the most popular software development tool ‘Visual Studio’ now also includes a graphical code visualizer called ‘Code map visualizer’.

**Code map visualizer**

Visual representation of software and information has been found to reduce cognitive load of a developer (Ball & Eick, 1996; Baker, Boilen, Goodrich, Tamassia, & Stibel, 1999; Purnell, Solman, & Sweller, 1991) by assisting cognition and comprehension of program structure and behavior. One such tool is Microsoft Visual Studio’s code map visualizer, this tool allows a developer to look into not only the code s/he is working on but also how the code fragment is related to other parts of code, practically displaying a live code flow, which allows much more educated programming. Microsoft developer network comments on code maps: “Code maps help you avoid getting lost in large code bases, unfamiliar code, or legacy code. For example, when you’re debugging, you might have to look at code across many files and projects. Code maps help you navigate around these pieces of code and see relationships between them” (Microsoft, 2014). Hands on training with other team members can be greatly improved with the help of visual information tools, for example:

- Understanding code dependencies.
- Understanding change impacts.
- Understanding how a piece of code integrates with the bigger picture.
- Educated debugging, that enhances system comprehension of an individual.

When a new person is in information gathering phase, such tools can ease and assist the information gathering process by not only letting the new entrant visually capture program information but also gaining insights into the ‘bigger picture’ of the software system. Experienced team members can write comments to the code map diagrams to further enhance the information relay process.
Some studies like (Purnell, Solman, & Sweller, 1991) have found that technical illustrations (visual representation of knowledge) have considerable cognitive advantages. This directly attributes clear advantages of comprehension and reduced cognitive loads on software developers who enter a new team and are posed with understanding a whole software system within a limited period of time.

Experimental Settings for Software Knowledge Sharing

Existing literature point towards efficiency of single room large screen information presentations for software development meetings (O'Reilly, Bustard, & Morrow, 2005) such meetings are also advocated by agile software methodologies. With smaller agile teams meetings conducted over a table mounted with display that presents required visualizations has high potential of boosting information visibility and creating awareness in agile teams.
3. Research Method

The empirical work for this thesis research is primarily done through participatory research and informal interviews and observations with a software organizations development team members. The goal was to gather perspectives, feedback and third person perspective on how well a developer can utilize available information available to him/her from other team members or documents. What are the factors that can influence this knowledge transfer process?

3.1 Participatory / Field Research

The research method for this thesis partially utilizes phases identified by Howell (Howell, 1972). These steps or phases as listed by Howell are:

- Establishing Rapport, the researcher establishes him or herself into the group under study as one of them
- Immersing oneself in the field, behaving and performing similar tasks as the group under study
- Recording data and observations, taking notes and/or interviews
- Consolidating the information gathered, compiling gathered data.

The quality of data recorded depends on how well the researcher has integrated him/herself and the degree of acceptance from the group. In my thesis case, I participated as a complete participant (see table 2).

During the course of my observations Company-A’s software development team structure went through restructuring; from a monolithic team to separate teams handling core development, UI team and the UX team (UX team became a shared resource for other product lines in the umbrella company which had acquired Company-A); data collection hence was restricted to the Core and UI teams. My participation level (see table 2) was ‘Complete Participation’ as I was inducted as a full time software designer in the organization. Software designer role handles development and design of software artefacts as well as maintenance of existing software code by debugging and fixing problems.

The observation data was collected while conducting my work of developing and debugging the software artefacts; as software development requires collaboration with other members, insights into their experiences was gained through routine daily work. During my work period of one year (Oct 2012 – Nov 2013) in Company-A, I recorded these observations as and when encountered either directly by me or through a co-worker. Also during informal meetings/interviews with co-workers I recorded observations as electronic or post-it notes, and later consolidate them into one
Each observation was then mapped to the categories listed in table 4 (Team Dynamics, Co-Ordination, SW-Development process, Technical Capacity). The categorization was made based on the nature of each task and the skills set each member involved possessed, members involved, location of the involved members; then observations were analyzed in the light of relevant literature.

Like any other qualitative research, there are ethical issues in my thesis work as well. As literature identifies that tensions can be spawned by research between research aim and participants’ rights to privacy (Orb, Eisenhauer, & Wynaden, 2001). These authors also raise concerns about how access is gained to the information collected, however as mentioned earlier all observations made for this thesis was collected as a full time member of the team and with the immediate team members and department manager’s intact awareness of the observations being made. The department manager shared his knowledge about the existing software processes but also often deferred information sharing requests to higher management who did not answer my queries, which demonstrated reluctance of making this information public. This is also the reason why the names of the individuals and organization is not disclosed in this thesis.

I have accumulated approximately 8 years of industry experience in various roles in different software organizations, and have experienced the significance of knowledge sharing and how software teams face knowledge sharing problems. Hence when joining this new team I noticed there is only casual knowledge sharing methods existing in the organization consisting of a wiki repository which included essentially only the requirements specification documents in textual form with scarce illustrations of the functionality if any at all.

Having worked in the software industry and knowing the different approaches software organizations opt for to share their knowledge, I chose the knowledge sharing area for my research work as it also coincided with my previous experience and current work in Company-A. I chose the topic as it would provide data thru my daily routine. Interest in this research area was only on department manager’s side possibly because most of the development team was experienced enough to not require or recognize the importance of knowledge sustenance measures. Qualitative research is prone to subjectivity (Ramos, 1989), in order to maximize objective analysis of observations only titles and designations are disclosed in this thesis, as this research was a critical analysis of knowledge sustenance in the organization. This is in line with references found in literature which highlight respect for the subjects of a research (Capron, 1989).

Most of the team has evolved with the product. Growing the knowledge depth as the product’s code grew to become a fairly large code base. This thesis will focus on gathering information on how the existing team have kept and retained their knowledge of the product. I chose knowledge sustenance research topic due to knowledge based nature of software development work and hence the key role that knowledge plays in software organizations specially the ones that resort to agile development methods where emphasis on planning/documenting is low. The analysis of gathered data was drawn by applying the principles and ideas found in literature in software engineering, information processing and cognitive sciences areas.

Knowledge sharing mechanism and its perceived effects as seen by management. How has the management influenced over time growth of the code base.

Management’s degree of ‘awareness’ and methods of delivering this knowledge to them i.e. verbose vs Graphical.
Types of Participatory Research

Participatory research can be carried out in different forms which differ from each other in terms of the process that defines them and the limitations each one of them incurs on the research outcome. Some authors have classified this research method into the following (DeWalt, DeWalt, & Wayland, 1998; Spradley, 1980; Schwartz & Schwartz, 1955)

<table>
<thead>
<tr>
<th>Type</th>
<th>Involvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-Participatory</td>
<td>No contact with population or field of study</td>
</tr>
<tr>
<td>Passive Participation</td>
<td>Researcher is only in the bystander role</td>
</tr>
<tr>
<td>Moderate Participation</td>
<td>Researcher maintains a balance between &quot;insider&quot; and &quot;outsider&quot; roles</td>
</tr>
<tr>
<td>Active Participation</td>
<td>Researcher becomes a member of the group by fully embracing skills and customs for the sake of complete comprehension</td>
</tr>
<tr>
<td>Complete Participation</td>
<td>Researcher is completely integrated in population of study beforehand (i.e. he or she is already a member of particular population studied).</td>
</tr>
</tbody>
</table>

There’s a risk that participants when being observed by a researcher may start acting or behaving different compared to their natural circumstances; as some authors state: Researchers engaging in this type of qualitative research method must be aware that participants may act differently or put up a facade that is in accordance to what they believe the researcher is studying (Douglas & Johnson, 1977). I believe this risk was non-existent in my observations because of my status as a full time team member with the development team. Being a full time permanent member of the development team, my observations were not considered intrusive or external. Nature of my work included close collaborative work with other team members, QA & other teams and to some extent with the management.

Because of my full time employment into the team, I was able to conduct the observations as a complete participant, taking notes during team discussions, informal meetings over lunch or coffee breaks and others such informal meetings. The department manager and my immediate team members was aware of my observations
however because of my complete participation recording observations was not abrasive to their daily activities or behavior. My research interest in this area was highlighted by the work practices and the state of software knowledge and its transfer in the company.

**Limitations of Participatory Research**

As mentioned earlier, participatory research involves a risk of change of behavior and/or actions of the subjects because of ‘awareness’. Besides this limitation there are a number of limitations depending on type of observation being utilized (see table 3). My observations are liable to be effected by loss of objectivity; while acting as a full member of the observed group.

<table>
<thead>
<tr>
<th>Type</th>
<th>Limitations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-Participatory</td>
<td>Unable to build rapport or ask questions as new information comes up. (DeWalt, DeWalt, &amp; Wayland, 1998); (Schwartz &amp; Schwartz, 1955);</td>
</tr>
<tr>
<td>Passive Participation</td>
<td>Limits ability to establish rapport and immersing oneself in the field. (DeWalt, DeWalt, &amp; Wayland, 1998);</td>
</tr>
<tr>
<td>Moderate Participation</td>
<td>This allows a good combination of involvement and necessary detachment to remain objective. (DeWalt, DeWalt, &amp; Wayland, 1998; Schwartz &amp; Schwartz, 1955)</td>
</tr>
<tr>
<td>Active Participation</td>
<td>This method permits the researcher to become more involved in the population. There is a risk of &quot;going native&quot; as the researcher strives for an in-depth understanding of the population studied. (DeWalt, DeWalt, &amp; Wayland, 1998)</td>
</tr>
<tr>
<td>Complete Participation</td>
<td>There is the risk of losing all levels of objectivity, thus risking what is analyzed and presented to the public. (DeWalt, DeWalt, &amp; Wayland, 1998);</td>
</tr>
</tbody>
</table>
4. About Product-A

Product-A development project was started in 2004, with a monolithic initial team that included software developers, graphics designer and user experience/user interface designers. One thing to note here is that the development team also included three subcontractors who were not part of the Company-A. Among these subcontractors only one worked onsite on need basis. Other two worked offsite, one in Croatia and the other in Australia. The target domain market for this project was initially set to be small and medium sized organizations especially in IT and marketing industries. Product-A targeted the SaaS market with its many benefits. It was built to provide its customers with a reliable, easy to use and efficient PSA (professional services automation) product, without the hassles of creating and maintaining local instance of software application. Professional Services Automation (PSA) is the underlying business process infrastructure providing a structured and standardized approach to the services delivery lifecycle. (PSVillage, 2007)

Entering the PSA (professional services automation) market with an affordable ready to use software service was very profitable for Company-A because of virtually complete absence of local competitors; there was however one very successful competitor based in Helsinki but unlikely to attempt a capture of Lappeenranta market. Product-A provided a promise of added value to its customers by streamlining the service and business processes.

4.1 Early Development

Initially Product-A was started as a development project based on Microsoft’s ASP technology using Visual Basic as the programming language, with Microsoft SQL Server as the applications data storage. User interface was built using HTML 4, CSS and scarce usage of JavaScript. Microsoft SourceSafe was used as source control provider. In early stages of business the software development process consisted of the following steps:

- Informal software requirements specifications.
- Informal software design.
- Implementation of the software.

Apparently no explicit method of knowledge preservation was followed. Documentation involved source code comments. Formal methods of recording and preserving software architecture and design documentation like class diagrams and other UML artefacts were not favoured. There was no software development model formally followed. Information gathered from current team members revealed that early work practices resembled the waterfall model with exceptions like interruption of development cycle of a certain module to re-assign the developer on some other task.

Product-A, despite its development cycle drawbacks was targeting a very fertile market segment and hence its market share was growing with a good rate. Relatively wider
customer segment started to request customized features for the software service to match their specialized requirements in their respective industries. Customer requests were contributing to the expansion of functional scope for product-A, hence increasing its applicability to the target markets.

With business growth like in any other organization the development team experienced growth as well. New members were integrated into the development team, some members were assigned new roles, while retaining team structure as single unit except the UI/UX (user interface and user experience) team, which was recognized as a separate team. Eventually part of the development work was outsourced to subcontractors who started developing from offshore locations.

4.2 Second Phase of Development

Eventually, starting from version 2 of product-A, Company-A moved its development platform from the existing older technology (Microsoft ASP) to Microsoft’s newer platform called ASP .NET. This platform had a variation of VB programming language called VB .NET. Team structure still composed of UI/UX team and development team, with both falling under the company’s R&D department. MSSQL server 2008 was retained as data provider. Currently product-A is in its version 3.x.

One of the product-A’s subcontractor was living in Croatia and another team member relocated to Australia and switched his role to also act as a subcontractor. The company had now decided to formally start following the SCRUM method of software development, with the intention of increasing the agility attributes of their development process. Holding daily scrum meetings in addition to per release retrospective meetings where the good and not so good attributes of the release work are identified.

Source control was also upgraded from Microsoft SourceSafe to Microsoft Team Foundation Server (TFS), which would later be replaced Atlassian JIRA.

4.3 Development Process

As mentioned earlier, Product-A development is carried out through SCRUM process. Team organization in this project has been more of a monolithic nature from the start until recently (spring of 2013) when two sub-teams were created.

These sub-teams were:

- Core team
- UI team
- UX team

Before the above mentioned team organization, major software design was done outside the company by one of the subcontractors, and none of that knowledge was brought in and retained within the company boundaries.

In addition to the sub-teams mentioned above Product-A team also includes the following:
Average day to day work for the development team is comprised of new features development and debugging existing source code. Product-A being a SaaS product, relies on new features development to enhance and expand its business. The process of identifying and building new features starts from product owner/managers feedback as well as comments and feedback from customer support team. This information is then compiled into a set of requirements specifications which are then handed over to the development teams. Development team then creates software design descriptions based on the specifications document and eventually implement them. The design descriptions are done in verbal form.

4.4 Knowledge Transfer to New Team Members

According to Cohen & Levinthal (Cohen & Levinthal, 1990) an organization’s capacity to learn new knowledge is dependent on the organization’s level of prior knowledge in the related area. In case of product-A, I have also noted that besides this capacity, the knowledge transfer efficiency also depends on the software development practices, the social structure and culture of the organization, however the focus of this thesis is on software development practices which is why the cultural aspects are not discussed in detail. The development process of Product-A was formulated such that there is minimal knowledge spill. Development is carried out in compartmentalized fashion where no single team understands the systems architecture completely. This could easily be remedied by delegating the software design tasks to designated software architect. Done in this manner the organization can ensure uniformity and standardized software design and development, hence making it easier for the team to discover and even infer new knowledge based on the fact that the system is uniformly designed.

Cohen and Levinthal (Cohen & Levinthal, 1990) also highlight that lack of investment in building the learning capacity will foreclose the future development of technical capability in that area. The following factors in case of Company-A can be seen to act in this direction:

- Personal goals are predominant instead of team culture
- Knowledge spills are minimal partly due to the above reason
- Knowledge innovation at its minimum

For an example of the lack of knowledge dissemination especially in terms of implicit knowledge let’s look at one of the development session of product-A in Company-A. This example illustrates how additional time costs are incurred when a software systems development infrastructure is not obvious. Incidents like ‘reinvention of wheel’ can hence occur. Actors involved are: A newly hired software designer, customer support representative and software architect. The software designer receives a task for multilingual translations addition into the system. Translations addition process is relatively long repetitive process but there’s no time for writing a tool for doing that.
The customer support representative accidentally notices this manual work and informs the developer about a tool that Company-A already possesses for this purpose.

Points of interest here are:

Information about the tool should have been known to the software architect already but it was not made known to the new member.

Information required for a certain task was not wholly represented in the request received by the developer i.e. it did not mention the tool that could assist in this task.

Factors that caused this outcome fall into a separate category of discussion falling out of scope for this thesis, but the effects that I’ve mentioned above highlight the lack of knowledge transfer where it immediately incurred more cost to the company by consuming more time.

The relationship between a new employee and organizations existing knowledge goes through a transformation as soon as the employee is inducted within a team. Once an employee becomes part of the team, s/he starts acquiring external knowledge. This knowledge is external to the individual but internal to the organization, but the individual is part of organization, so the knowledge acquisition can also be seen as a circular knowledge absorption for the organization. Thus Cohen & Levinthal’s observations about knowledge absorption and the factors that apply can also be applied here; according to their research organizations learning (absorption capacity) depends on the capacities of its individual members. They have mentioned the term absorption capacity in both existing and new knowledge creation contexts as well as the capability of the organization to utilize the knowledge created or acquired. In the example given earlier for case Company-A, they had successfully retained a piece of information within the organization but were not able to utilize it effectively. Research on memory development suggests that accumulated prior knowledge increases both the ability to put new knowledge into memory, what we would refer to as the acquisition of knowledge, and the ability to recall and use it. (Cohen & Levinthal, 1990)
5. Utility of knowledge management

This chapter discusses the observations collected in Company-A and their effects on knowledge creation and effective transfer to new entrants in a software organization. Knowledge management strategies that are a combination of a dominant and supporting strategies are found to be more successful (Hansen, Nohria, & Tierney, 1999). Hansen et al. state an 80% dominant - 20% supporting relationship in this strategy recipe. According to them attempting to excel at both strategies results in failure.

Mutual trust and commitment to a company as identified by Nonaka (Nonaka, 1994) plays very significant role in knowledge creation. Every individual who has been contributing to significant parts of a software system should become a vital contributor to the knowledge base (personalization or codification from Hansen et al, 1999). Knowledge base for a software organization includes but not limited to individuals, wiki articles, source code repositories, bug reports, resolution reports, requirements specifications, design plans and user stories etc. Knowledge contribution requires incentives; which according to Hansen et al., should be real incentives – not small enticements (Hansen, Nohria, & Tierney, 1999).

5.1 Observations

Based on analysis of extant literature in fields of software engineering, information systems and social sciences and the observations conducted as part of my employment I have come to a conclusion that the software development process and the resultant knowledge creation depends on not only the execution of core software development process but also the its related socio-technical factors, following is a list of categories my observations can be classified into.

- Team Dynamics (cases where team structure hinders knowledge management)
- Co-ordination (cases where co-ordination practices can be improved)
- Software Development Process (cases where development process can be improved)
- Technical Capacity (cases where team’s tech capacity creates knowledge obstacles)

In this section I am tabulating a list of vital observations that I have collected through my participatory research, conducted during my employment as a software designer in Company-A. Table 4 lists these observations, where each row represents occurrence of a certain incident (task), team member(s) involved (assignee), scope of change and the outcome caused in context of knowledge creation, sustenance or transfer. I will elaborate each case further following the table in order to create a clear understanding of the theoretical aspects that will be applied to each case.

Note: table entries are color coded to highlight the category(s) they fall in.
<table>
<thead>
<tr>
<th>Task</th>
<th>Assignee</th>
<th>Scope</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Reporting UI</td>
<td>Subcontractor in Australia</td>
<td>Build JavaScript based user interface mechanics for reporting features of the application</td>
<td>Built and handed over in incomplete state to a new employee without any design documents. New developer takes a few days and accidental help from another new employee who pointed out that the JavaScript being used is distributed online and not built internally. After which it was easier to find required information online.</td>
</tr>
<tr>
<td>2. Hour entry delimitation of 24 hours</td>
<td>On site team</td>
<td>Modify hour entry feature to accommodate 24 hr. format.</td>
<td>Technical limitation based on back end where one day = 00:00 to 23:59 hrs. Without giving due consideration to this, architect insists on creating work around to accommodate the change and hence creating more dependency on specialized knowledge instead of standardized knowledge.</td>
</tr>
<tr>
<td>3. Modifications in currency rounding methods</td>
<td>On site developer</td>
<td>Modify existing currency rounding methods to include rounding rules from different countries</td>
<td>Feature dismissed from the release due to incomplete information, release could have led to legal issues as some countries do not allow rounding certain values and prefer precision. The developer had finished entire new implementation when it was dropped. Boehm calls this knowledge asymmetry (Boehm, A view of 20th and 21st century software engineering, 2006)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>4. KPI Implementations</td>
<td>Subcontractor</td>
<td>System Database to display KPI(s)</td>
<td>Database table used in code as aliased table. No one else on team knew the reasons why the implementation was using aliased name and hence did not have enough information needed to rectify the situation, which meant the fix had to wait until the subcontractor could fix it.</td>
</tr>
<tr>
<td>5. Report display auto fit for screen width</td>
<td>On site developer</td>
<td>All reports in the system</td>
<td>Customer service requests the feature, implementation is done. Software architect is not completely aware of the implementation, developers find and fix the problem resulting in one report not operating properly, software architect mandates roll back of all changes being unaware of the scope of change, seeing the roll back as the only possible rectification.</td>
</tr>
<tr>
<td>6. Physical File database</td>
<td>On site developer</td>
<td>File storage and retrieval</td>
<td>Feature requires file storage and retrieval for system users, on site team spends considerable time figuring out that the project contains a second database meant only for file storage. This information was not readily available to the team.</td>
</tr>
<tr>
<td>7. Unintended product release</td>
<td>On site QA team</td>
<td>Scheduled product release</td>
<td>During a scheduled feature development and debugging of the product, Core team leader decides to not release the product on scheduled date, but this information is not properly relayed and QA carries out the release.</td>
</tr>
<tr>
<td>8. PSA, Admin Application, Extranet</td>
<td>On site team and subcontractor</td>
<td>Different components of the PSA software system</td>
<td>Three modules of the PSA project contain identical functional components and have redundant dependencies which are not widely known.</td>
</tr>
<tr>
<td>9. Code reviews</td>
<td>On site team and subcontractor</td>
<td>All source code changes</td>
<td>Source code reviews are done by development team and subcontractor but instead of change requests to the original developer, changes are directly made by the reviewer without any information relayed back regarding the reasons for said changes.</td>
</tr>
<tr>
<td>10. Translations implementation</td>
<td>On site team</td>
<td>All source code changes</td>
<td>PSA project supports multilingual UI, achieved by storing translations in a database, the implementation of this functionality is not uniform in some parts of the project and is not clearly known by the team.</td>
</tr>
<tr>
<td>11. Graphical Reports (Gantt) implementation</td>
<td>On site team</td>
<td>Reporting UI</td>
<td>Gantt chart in PSA is implemented in two different ways (implementation A &amp; B). A reported bug in implementation A falls out of scope of local team and should be delegated to the component vendor. This technical understanding is not available to QA and thus QA keeps the bug in ‘active’ state.</td>
</tr>
<tr>
<td>12. Management Personnel/Sales Leaves company</td>
<td>On site team</td>
<td>team wide change</td>
<td>Person responsible for organizing and approving release content and interacting with customers as well, leaves the company without properly planned knowledge transfer procedures are carried out.</td>
</tr>
<tr>
<td>13. Date picker implementation change</td>
<td>On site team (UI)</td>
<td>Application wide change</td>
<td>UI team is designated to create an alternative date picker implementation based on jQuery date picker. Off-site subcontractor and UI team lead starts developing with non-standard methods of localization. Decision</td>
</tr>
</tbody>
</table>
14. IoC implementation

On site subcontractor starts developing IoC implementation. Knowledge about this implementation is highly compartmentalized. Request for training is not met with much interest.

**Reporting UI**

This task was started when designated Software architect and one of the subcontractors decided to renew existing reporting functionality with new features, in order to fuel customer interest and utility value. The subcontractor is based in Australia and visits the central office in Finland bi-annually. All communications are Skype based. New features are based on a JavaScript plug-in available under open source licenses. Information about usage of the plugin is not readily available. JavaScript modifications done in the plug-in have syntax errors.

Later on some tasks are delegated a newly inducted developer A. Developer A has to rely on electronic communications for every question, communication is slightly inconvenient because of time zone difference. However accidental input from another new team member points out that the JavaScript being used is distributed online and has not been written by the subcontractor. This information opens alternative information sources for developer-A (internet based forums, blogs and other sources) because JavaScript source code was freely distributed and maintained by the original author.

Analysis: Vital contextual information was missing from the task, resulting in cognitive overload for the new team member and loss of efficiency. This observation highlights that in order to properly delegate this task, transferring source code only was not sufficient. Developer needed to resolve a number of questions. Appropriate information was not even present at the sub-contractor’s end, as the JavaScript plug-in was not originally developed by the subcontractor. Visualizing a simple Dependency Graph would have provided both references to the new developer as it would have identified an external dependency on the JavaScript plug-in as well as improved understanding of internal dependencies of the source code.

This occurrence highlights that lack of proper coordination between the subcontractor and onsite developer led to additional time spent on the task. The method of knowledge transfer used by the subcontractor included transferring source code only and online communications via Skype, where both the subcontractor and the system architect failed to identify and relay the information about external dependencies of code to the developer. As mentioned earlier, a standard dependency graph along with the source code would have pointed the developer in right direction, as well as preserving a design document which will provide necessary information in an easier to comprehend, standardized format for new comers.
24hr Format of Hour Entry

Product-A contains a feature for time entry. This feature is called hour entry form and originally did not allow input in 24 hours format. As a new feature 24 Hour format for hour entry was decided. There was no prior software design done and the task was handed over to the development team (myself and one more developer) in the form of requirements specifications text. The underlying database for product-A is MS SQL server which natively allows 24 hour date time format, but the starting standard values for hours in this format range from 00:00-23:59, however the specifications were based on 1-24 format which from time standards counts as 25 hours. Because of this specification another work around had to be done to offset the automatic addition of 1 day to the hour entry whenever someone enters 24 as input value (since 24 > 23:59 MS SQL server treated that as a value spanning two days.

Analysis: This situation demonstrates technical capacity deficiency, which led to a non-standard implementation of a software feature. Firstly the software design approach which was utilized to develop the feature, lacked proper insights; eventually resulted in additional complexity in the source code i.e. add 24 hour entry capability to the hour entry form and then create additional logic to circumvent MS SQL Server’s handling of hour value of 24.

Knowledge regarding product-A’s infrastructure capabilities was not known to the system architect, as no mention of the additional logic for storing 24 as two days was mentioned in the textual specifications. Further down in the application life cycle several issues arose regarding this functionality because of mismatch of the application and infrastructure handling of 24 hours values.

This observation was included because it highlights the effects a team’s existing technical capacity can cast on knowledge transfer. Smooth knowledge transfer for new team members requires that the knowledge is easily understandable and logically structured. However, incidents like these create extra learning curve and ‘anticipatory exploration’ can actually turn negative and inefficient. One also note that the decisions about work-around(s) were made on ad-hoc basis as the specifications document did not identify the need for a work around. This valuable contextual information will be lost as soon as either the designer or the architect is no longer available for the new team members.

In order to easily store this information for future use by existing or new employees, a technical feature design document (class, activity, use case, flow, and state) would transfer more information in the least taxing manner for a learner.

Modifications in Currency Rounding Feature

Product-A’s currency rounding logic used to be uniform for all countries where its customers are based. Later on a requirement was identified for customers in Sweden regarding the way currency rounding is handled. Requirements for these country specific currency rounding was written in the conventional textual form and handed over to a developer. No prior software design was carried out, but the developer got some help from fellow team members to come up with a software design for the said feature.
Development of the feature was completed, QA testing was also carried out and the feature was considered ready to be delivered, when it was identified that the currency rounding in the system and on the generated invoices which already exist should not differ at all (as the new rounding only applied to new invoices). This information was not present in specifications document and the legal context was not known to the development team; consequently the feature had to be completely removed from the release version of product-A, and had to be entirely redone.

Analysis: Incompleteness of information here is two pronged. First, if a technical design was done by the architect instead of relaying textual specifications to the developer, the logical dependencies of the feature would have become known prior to the development. Secondly, the implementation knowledge of existing source code for rounding was held by off-site subcontractor and in textual form in the wiki; and did not fully educate the developer or the developer on his personal level did not invest enough time reading through the verbose description of the functionality, in either of the cases it is clear that the information was not easily and readily available, creating a knowledge gap. Boehm calls this phenomenon knowledge asymmetry (Boehm, 2006) i.e. knowledge existed but in high degree in one part of the team and low degree in another, there was lack of transfer of this knowledge resulting in an inefficient outcome. Hence this incidence can be classified into co-ordination and development process flaws which hinder knowledge transfer. Moreover even though the problem was identified before it went out to production servers, knowledge regarding this was not recorded for future team members. Technical design documents of the feature still did not exist and major part of required knowledge was held by the off-site subcontractor (re-implementation of the entire feature was carried out by the subcontractor). In this case software development process and co-ordination pattern dictated compartmentalization of knowledge.

**KPI Implementations**

Product-A provides a feature called KPI (key performance indicators) which are both system defined and user defined calculations (formulae) for representing useful business information for the user. Implementation for these KPI(s) has mostly been done by an off-site subcontractor. Major parts of the logic are not clearly understood by on site development teams; a good example of maverick programmer scenarios as identified by (Ravichandran & Rai, 2003). During one of the releases, some feature extensions were required to be made to the KPI, that resulted in an exception displayed on user interface (product-A’s ‘Dashboard’). Since there was no design document (except a requirements specification document which does not cover technical details of the implementation) and immediate access to the subcontractor was not possible, the on-site team’s only way was to start digging into the source code, it took a while before the cause of exception was discovered. In Company-A in another product line, similar functionality had been implemented in a much simpler way yet the knowledge available there was not utilized to simplify product-A’s implementation of KPI(s), yet again due to the scale of compartmentalized knowledge.

Analysis: In this scenario product-A’s system Dashboard was programmed to display KPI(s). KPI logic was split in the code and database. It was later discovered after considerable code digging that the implementation is using the related database table as aliased table name. No one else on team knew the reasons why the implementation was
using aliased name and hence did not have enough information needed to rectify the situation, which meant the fix had to wait until the subcontractor could fix it.

Research identifies software development being always dependent on experts (Blackler, 1995); on the other hand it is also noted that modern software development requires more knowledge compared to software developed earlier (Ye, 2006). The case of KPI(s) portrayed presence of knowledge asymmetry (Boehm, 2006) and demonstrated the inefficiency incurred because of it. This asymmetry created knowledge transfer problems; scope of co-ordination among the onsite developers was limited also because of absence of appropriate technical designs. Knowledge asymmetry, team dynamics and co-ordination. All three have played their role here. Team dynamics is a wide ranged topic but here I am referring to the sections of this topic where, team members start owning a source code to a degree where they see co-ordination, co-development as negative influencers. Alternate opinions are disregarded (i.e. second product line’s implementation of KPI features), comparative analysis of implementation methods and software design practically do not exist. Combination of the factors above hinder knowledge sustenance in easy accessible forms because the need for such assets is overlooked as not needed since the maverick programmer (Ravichandran & Rai, 2003) is still present and working.

Report Display Auto-Fit

A very useful feature of Product-A is ‘Reporting’. Users can create, save and share a number of reports that display user data in different formats providing several perspectives over business and operations. The reporting feature, as with KPI(s) has been largely developed by an off-site subcontractor. All major development is delegated to the subcontractor. Knowledge transfer in this scenario is very difficult. The user interface of these reports had historically been designed to occupy about 75 percent of the screen real estate, but provided a ‘maximize’ button which was supposed to maximize the report to available screen size; this functionality did not work as intended and hence a request for fix was made by customer service department.

Analysis: Co-ordination between the customer service and system architect was not carried out properly. Development was interrupted due to confusion about who has requested this bug fix and should it be done at all. Since the request did not come through the system architect there was considerable opposition in carrying out the fix; subjectivity came in to play and the whole fix was eventually rolled back. A potential solution was not documented and the knowledge acquired and created in this scenario only exists as individual memories. Software architect was not completely aware of the technical implementation, hence a knowledge gap was created without any rectifications. Such knowledge gaps can potentially cause unnecessary time investments in future with new members who will only find required knowledge by digging through source code.

Physical File Database

Product-A allows users to upload files for storage and retrieval. The functionality is built as one of the core features of the system. Implementation of this feature utilizes a
separate database apart from the main database for the whole system. This file database is only used for keeping file records.

After organizing development team into ‘Core’ and ‘UI’ teams the most experienced (based on time spend in Company-A) members were allocated in one team and UI team mostly contained new members. No one in the UI team and most members of the ‘core’ team were not aware of the separate file database existence. One of the individuals who worked on this feature was no longer present in the company and the other one was the off-site subcontractor. Discovering the true structure of this functionality incurred significant time cost for the development team.

Some team members in Company-A advocate ‘Asking when needed’ practice; which in my experience has been very unproductive as in real life situations, especially in software development teams, individuals are always doing knowledge intensive work, breaking someone’s thought chain is neither productive nor very likeable by the person being approached. Older team members have expressed their unhappiness during day to day queries being directed to them, stating that it breaks their thought chain and takes a while for them to reconnect their minds back.

Analysis: This is a classic example of team members leaving and new members joining the company and in course of this turn over loosing vital knowledge. Documents at the disposal of development team consisted of wiki entries and specifications, which were verbose and were subject to ‘interpretations’. Moreover there are studies like (Johansson, Hall, & Coquard, 1999) stating that written material is rarely consulted by real life teams in real life situations. Team members look for the quickest possible way to accomplish something and ironically end up spending more time trying to understand the problem then to solve it. In this scenario of product-A’s file database a proper technical design document illustrating the software modules and its dependencies would have quiet quickly and easily portrayed the required knowledge to new team members, transferring the knowledge in a graphical manner depicting the system is using two databases. In addition to quickly bring the new member up to speed a technical document can also pave the way for anticipatory exploration for further and deeper understanding hence contributing also to the notion of agility.

**PSA, Admin, Extranet**

The structure of Product-A is such that a set of functionalities exist in separate applications called Extranet and Admin Application. Extranet was built for content sharing created by Product-A. Admin application, as the name implies provides administrative access for product-A. Software design of the three applications, is so that there are a number of software modules utilized by all three and have three individual implementations, though some degree of reuse exists; much more of the existing functionality could be commonly built and shared by all three with a single code base.

Analysis: Most of the onsite team does not worry about code duplication or is either unaware of its existence. The knowledge about what parts of source code are actually duplicated is not readily available anywhere except the code itself and requires considerable time for analytical discovery. This knowledge must have existed at least at the time of creation of these redundancies but was lost eventually due to employee turnover. This state of a software system indicates inefficient software design and goes
unnoticed because the knowledge of its existence does not exist and hence is less likely to be highlighted or propagated.

**Code Reviews**

In Company-A as measure for control source code quality, code reviews are carried out by designated team members. A good practice that it is, if carried out improperly loses some of the long term benefits for the development team and the knowledge of the organization as a whole. Some reviews are carried out by offsite subcontractors as well and provided no knowledge creation value.

Analysis: Code reviews that mandated any code modifications were not delegated back to the developer with modifications highlighted. Instead, the intended modifications were made to the source code without providing feedback to the original developer. Because of these no-feedback changes, the developer’s knowledge level is deprived of an obvious improvement. This behavioral pattern might be rooted in the fact that individuals try to shield themselves from their environments (Guinan, Cooprider, & Faraj, 1998). Driving factors behind these incidents could range from perceived notions of losing control, getting left behind by technological advances or competitiveness among team members.

Code reviews are a very good way of disseminating knowledge if utilized properly i.e. all code reviews provide change requests and brief descriptions (if done by an offsite personnel) about the modifications requested. In this manner a developer carrying out the requested changes will not only understand the problem at hand but will also draw information about the holistic picture of the software system.

**Translations Implementation**

Multi-language support is provided out of the box by product-A. From the technical perspective product-A achieves this by keeping phrases from all supported languages in a database and pulling out the required phrase sets. These phrases are called translations and developers (including myself) are required to update the database and source code every time a new statement, word or label is added to product-A. The process for this is present in two forms, based on what part of the application is the phrase being added to.

Analysis: Two different processes of adding translated phrases to the system exist without a clear distinguishing factor, making it especially difficult for new team members to grasp. Team coordination cannot cover all technical implications unless and until a case which directly touches a certain section of software is handled. Knowledge transfer is best done hands on (Johansson, Hall, & Coquard, 1999). In cases of new team members, the practice of paired programming (new member paired with experienced member) can efficiently transfer such knowledge.

**Graphical Reports (Gantt) Implementation**

Reports based on user data is a highly demanded feature of Product-A. Some of these reports are graphical, for instance the Gantt chart. Product-A’s implementation of the
Gantt chart utilized FusionCharts component for Gantt graphical part. However within product-A there are two different implementations that use the FusionCharts component. One implementation started experiencing a bug. Upon investigation it is revealed that product-A contains two different implementations to display the Gantt chart. Further study led the team to realize that the problem cannot be solved without any input from the vendor i.e. FusionCharts, and hence the bug was put on hold with this information added as notes.

Analysis: Most team members were not aware of existence of two different implementations of the same functionality, documentation available to the team did not provide any insights into the technical details of the implementation. This lack of knowledge and information manifested itself as a QA-Development cycle where the QA engineer was not aware of disparate implementations and did not understand notes added by developer, consequently the QA-Engineer kept bouncing the bug back to developer (myself). This shows the importance of knowledge capacities of all teams directly involved with the development, not only developers but QA should also be capable of understanding technical specifications, descriptions and graphical representations in order to ensure that existing knowledge in a software company can be effectively utilized.

Date Picker Implementation

Product-A contains many user interface elements like text boxes, picture elements etc and one of them is the date picker / calendar element (DP). The DP implementation is old and so the UI team decided to update with a jQuery implementation of the same functionality. The jQuery calendar claims to be faster with a smaller footprint. As mentioned earlier product-A is available in different languages and locales; hence the calendar has to support multiple formats and languages as well. The platform, product-A is built upon (Microsoft .NET/ASP.NET) provides out of the box support for multi-lingual and multi-locale applications but the offsite-subcontractor and UI team decided to implement a custom method of localizing the jQuery calendar. In doing so they create more code to maintain instead of utilizing .NET platform’s standard methods of localization.

Analysis: Non-standard implementation of a feature (localization) created an extra layer of complexity for individuals attempting to understand the inner workings of the calendar implementation. Knowledge about the implementation details was not disseminated to the team, marking inappropriate coordination. Another member of the development team not directly involved in development of the feature brought the easier, standard method of localization provided by .NET but the development phase had progressed far enough to not allow re-development. The process of developing this feature and the choice of using non-standard method highlights that the technical capacity of a team can simplify or complicate knowledge transfer. As new team members will not be able to use any degree of ‘anticipatory exploration’ in this case or utilize knowledge resources available for the .NET platform regarding localization because the implementation is completely non-standard. This example highlights that the foundations of knowledge transfer capacity of an organization is also rooted in their social and technical capacities.
IoC Implementation

Inversion of control (IoC) and dependency injection are methods to create loosely bound software modules. In product-A’s case the onsite subcontractor started shifting the programming practices of the team towards IoC. Even though Product-A had been using IoC containers before, knowledge about this practice/technique was absent except for the onsite subcontractor. The team did not follow any properly planned course of including IoC on a wider scale, nor was the knowledge disseminated after initial development was done.

Analysis: In context of knowledge creation and sustenance, inclusion of IoC did not contribute. The already existing code that was utilizing an IoC container was unknown territory to most of the team members; this situation did not drastically change even with the new IoC source code inclusion. One of the UI team members (myself) requested a formal knowledge sharing session regarding the IoC only, but it was never carried out. The team’s co-ordination culture was formulated to only coordinate about what’s necessary for development at hand. With only one person (onsite subcontractor) available with clear understanding of the new IoC implementations, no technical documents there is huge risk of loss of valuable knowledge. IoC is not the simplest of concepts to pick up; moreover when used in a haphazardly developed system like product-A.

For effective knowledge transfer entire team should develop a culture of knowledge acquisition whenever and wherever possible. As noted in this incident, knowledge dissemination was not carried out even upon request. The perception of having mavericks (Ravichandran & Rai, 2003) influences work culture making the importance of awareness and knowledge dissemination appear lower. Coordination of the development teams should cross boundaries of features being developed so everyone can understand and explain parts of the system to new team members.
6. Discussion

Prior studies like (Johansson, Hall, & Coquard, 1999) argue that presence of formal communication methods can negatively affect face to face communications; which means the material present in written form should transfer knowledge very well to compensate for the reduced emphasis on face to face communications. My understanding of this concept is a bit different i.e. these two methods of communications should complement each other instead. A written document should preserve knowledge to the best possible level so that when a person to person communication ensues the documents can provide valuable contextual material. This combination of both methods can improve the transfer of both explicit and tacit knowledge. Face to face communications provide experience sharing with team members but because of complex nature of software may not completely provide a whole picture to a new developer (team member).

Along the same line of thought, regarding knowledge and experience transfer, Ackerman and Halverson argued (Ackerman & Halverson, 1998) that experiences are valuable only when they can be contextualized, decontextualized, and re-contextualized at the proper time, one has to be aware that mere existence of experience holding team members is not a guarantee of good knowledge transfer. Cognitive load (Sweller, 1988) and contextualizing software knowledge is much easier if represented in an easy to read and comprehensible form; and graphical knowledge representation forms are found to be more efficient in assisting comprehension and reducing cognitive load. (Purnell, Solman, & Sweller, 1991; Ellis, Wahid, Danis, & Kellogg, 2007; Storey, D., & German, 2005; Larkin & Simon, 1987). Therefore, for instance, a new team member might find it easier to understand a code flow diagram or an architectural dependence chart compared with large textual description of the system. Such documents can also provide valuable grounds to start training a new team member in distributed teams.

Prior studies in software engineering state that experience and knowledge is most effectively transferred with face to face interactions (D.Latoza, Venolia, & DeLine, 2006). In face to face interactions it is not only the problem at hand that can be addressed but spontaneous responses may also shed light on related tacit and contextual knowledge. However when it comes to comprehension of a software system comprised of complex dependencies and several levels of code calls, visual knowledge representation should be utilized accompanied with other forms knowledge transfer. Thus knowledge transfer creates a dependence on a number of factors like how complex is the software, is there a knowledge broker or software historian (D.Latoza, Venolia, & DeLine, 2006) present to guide him/her to a correct person. If software design and code is written so that only the person who wrote it can explain it, efficiency can further drop because in absence of the person who can explain code a developer is essentially blocked as mentioned by Perry et, al. (Perry, Staudenmayer, & Votta, 1994).
Software knowledge, tacit knowledge can refer to knowledge of the intricacies of a software system architecture, design decisions, architectural decisions or other Problem-solution space attributes. Figure 2 illustrates the possible outcomes of agility, when software is built on evolutionary basis, with little or no planning it often results in bad design (Fowler, Is Design Dead ?, 2014). Lack of planning in software design, creates tacit knowledge; since knowledge is personal attribute of individuals it generates dependency for all other members of the team. Externalizing or transforming this knowledge into explicit knowledge should be effective. Prior work in this research area shows that explicit concepts can, for example, be created through dialog using metaphors and analogies (Nonaka & Takeuchi, 1995); metaphors and analogies are attributes represented very effectively with standardized graphical forms like UML.

6.1 Agility and Knowledge Creation

Real-world examples argue for and against agile methods (in terms of their efficacy). Responding to change has been cited as the critical technical success factor in the Internet browser battle between Microsoft and Netscape. But over-responding to change has been cited as the source of many software disasters, such as the $3 billion overrun over run of the US Federal Aviation Administration's Advanced Automation System for national air traffic control (Boehm, 2002). Boehm also recognizes that in certain situations a hybrid approach combining traditional and agile methodologies can perform better than either. Software organizations often misinterpret the agile manifesto’s emphasis on planning than documents, as its dismissal of planning which includes technical planning (architecting a software product); thus landing in haphazard hit and miss approach for creating software is adopted; eventually creating serious knowledge gaps. Literature analysis and work observations point towards the fact that agility should not affect knowledge creation. Software organizations that claim to be agile
should clearly separate knowledge creation from core software development models to get rid of opinions that documentations are detrimental to agility.

Observation 4, 6, 9 and 10 show the direct influences of agility on a software development team’s knowledge creation capacity. 4 and 6 also contain team dynamics influences, that result from social aspects like goal alignment, politics etc. Since agility emphasizes on quick to market notions, software teams spent less time planning and more time getting a product or feature out. In 4, 6 the required features were built without valuable technical documentations. Moreover, they were built by team members who no longer work on site. This provides a very potent example of a team capable of churning out functional software artefacts swiftly cannot be guaranteed to carry that streak on for any period of time. Organizations change, people move and in software organizations following strict agile principles (acting over planning) knowledge moves away with people.

Some authors’ opinions on agility vs traditional models have also argued that agility is only meant for seasoned teams: “There are only so many Kent Becks in the world to lead the team. All of the agile methods put a premium on having premium people….“ (van Deursen, 2001). Boehm also argues that a few designers working together can work better than on their own only when they are ‘premium’ designers and adds that 49.9999% of world’s software developers are below average (Boehm, Get ready for agile methods, with care, 2002).

The nature of agile methods and the need to be agile creates high dependence on tacit knowledge (Royce, 1998) which can lead to knowledge gaps i.e. in face of employee turnover because tacit knowledge is carried by humans. Boehm (Boehm, Get ready for agile methods, with care, 2002) goes on to highlight the shortcomings of tacit knowledge reliance in agile methodologies by stating: ‘When the team’s tacit knowledge is sufficient for the application’s life-cycle needs, things work fine. But there is also the risk that the team will make irrecoverable architectural mistakes because of unrecognized shortfalls in its tacit knowledge’. Without externalizing significant knowledge, agile methods can fall short when it comes to knowledge transfer as the original employees might not be around to bring new team members up to speed.

An empirical study concludes that 80% of rework required in the analyzed project was caused by ‘architecture-breakers’ i.e. architecture discontinuities to accommodate performance, fault-tolerance, or security problems, in which no amount of refactoring could put Humpty Dumpty back together again (Boehm, Get ready for agile methods, with care, 2002). Such outcomes are caused by lack of foresight and planning in favor of agility; besides incurring costs such incidents also cause continuous addition of complexity into the system hence adding to cognitive load.

In traditional software development methods like the waterfall model, knowledge sustenance was not really an issue since the process itself produced documents as development assets. Agile methodologies on the other hand advocated more action and less overhead, agile software processes like Extreme Programming, Scrum and others, rely on direct face-to-face communication between customers and developers for knowledge sharing (Chau & Maurer, 2004). Though agile methods solve short term knowledge sharing problems, but when it comes to creating knowledge sustenance, it does not solve much. As seen in the observations at Company-A there were sections of software design and source code whose knowledge was present in the team, was either partially present, was misunderstood or was present with sub-contractors. Though
knowledge assets like requirements specifications, source code, wiki-entries were all present but their contribution for software knowledge was inefficiently minimal.

Knowledge sustenance is something that has to be sought after by the entire team. An individual trying to create a practice of recording and transferring knowledge will not be able to initiate much of a difference. Most successful changes, and especially a change to an agile process like Scrum, must include elements of both top-down and bottom-up change (Cohn, 2009).

The lowered emphasis on documents in agile methodologies is often misunderstood as an indicator of ‘insignificance’ of documents for a software organization.

Figure 2 illustrates the DUCA process, where an individual can begin by discovering what to explore and understand, then understand the phenomenon hence internalizing the knowledge found and then act upon a certain procedure where s/he can utilize the knowledge. In cases of software source code, where design and code writing is non-conforming to industry standards or predominant design patterns, the discovery and understanding phase can incur huge cognitive overloads for the knowledge seeker. Establishing a corporate culture that includes a coordinating mechanism for communication (to transfer knowledge and experience) is essential for employees’ knowledge-creating activities (Nonaka & Takeuchi, 1995).

Considering Company-A’s agile based software development process, distributed teams and absence of appropriate externalized technical knowledge the DUCA model will indirectly mandate significant effort on collaboration right from the discovery phase. Software developers, upon failing to find information needed, just defer their work until they can find a person who can explain and answer their queries (Ko, DeLine, & Venolia, 2007); because spontaneous availability of team members in distributed teams cannot be guaranteed a higher chance of deferring exists. Company-A
Effective knowledge sharing can circumvent deferred knowledge transfer problems, by creating an architectural documentation phase where high level technical illustrations are created. These documents can provide contextual information to developers, enabling them to start inferring, exploring and understanding the source code and conduct educated distance collaboration when needed.

6.2 Effects of Distributed Team Structure & Agility

Ravichandran & Rai point out very accurately that many software organizations rely primarily on a maverick programmer (Ravichandran & Rai, 2003); which creates knowledge sharing problems. Company-A’s work culture very closely resembled this statement. More over the knowledge sharing problems were further pronounced because the ‘maverik programmer’ in this case was a subcontractor and usually did not work on site. Instant face to face knowledge sharing thus was a rare occurrence. An empirical study by Seaman & Basili (Seaman & Basili, 1998) states that local mobility facilitates awareness in ways that are unavailable in distributed situations. Awareness can refer to both the knowledge of a software system and knowledge of others activities effecting the software knowledge of the organization. Another study touching on distributed teams and awareness states that lack of spontaneous communications can create inter-site coordination problems (Gutwin, R. Penner, & Schneider, 2004). Collaborating with team members for knowledge transfer therefore cannot be a smooth journey in such work environments.

Some authors argue that it is impossible to transfer knowledge precisely, but if the knowledge is merged with experience(s), it will most probably enhance competence (Johansson, Hall, & Coquard, 1999). Where experience is defined as capacity to act. In case of distributed team of Company-A, transfer of knowledge and experience faces more problems as hands-on timely transfer from one person is not always possible. Software source code can often be in state that unassisted comprehension poses a challenge as well, similarly other sources of information like manuals or wikis would pose the same problem. Very often, written and stored information is barely recognizable to outsiders, so the author usually has to be contacted to understand the context of the experience (Orr, 1996). When it comes to distributed teams where knowledge assets (documents, source code etc.) exist in long verbose forms the amount of cognitive overload exceeds any efficient knowledge transfers. Figure 3 below illustrates how knowledge assets, software design itself and visual representation together can circumvent effects of distance on knowledge transfer by providing contextual basis of discussions, exploration and explanations.

Figure 3. Precursors to electronic knowledge transfer. Source: Original research.
The above illustration (figure 3) outlines how appropriate software knowledge assets combined with good software design and represented with illustrated visual documents can facilitate collaborative comprehension for distributed agile teams. Research work done by Johansson, et al., identifies that individual interpretations of written work are very likely to be subjective. Johansson et al., (Johansson, Hall, & Coquard, 1999) point out to human tendency of interpreting information in subjective manner by referring to the postman game, where information is passed on from a person to another; the information received by the last person in this chain is very likely to be different from the original. Based on this argument they propose that the most valuable experiences are transferred by the actual person who gained it him/herself.

Observations 1, 3 and 9 portray the how co-ordination and knowledge sharing/transfer can get effected by distributed teams. Observation 1, also portrays team dynamics of misaligned individual goals, as faults pointed out by a new developer were seen as misinformed analysis. Such incidents defy Nonaka’s argument that every contributor in a team is important and his/her contribution valuable (Nonaka, 1994). Company-A’s development team has a designated software architect, but there is no value seen in sustaining documentation repositories of software architecture as knowledge. Application design work is done by the across the border (Croatia) sub-contractor, who has also attained the status of ‘maverick’ programmer of the organization. Available information on software design and architecture is mostly only comprised of verbose descriptions of requirements specifications and wiki articles. Textual functional specifications often lead to misunderstandings and misinterpretations. When most of system design experience exists outside the organization; and difficult to access knowledge on site, quick information and experience sharing is challenging.

Software businesses being knowledge based draw all value from people’s knowledge and experiences. Usually software systems are built by collective effort of people. The real value of information systems is connecting people to people, so they can share their expertise and knowledge on the spot. (Johansson, Hall, & Coquard, 1999). An empirical study carried out at Microsoft concludes that electronic communications like email, IM, message boards etc. do not make up for the entire value lost because of absence of face to face communications (Ko, DeLine, & Venolia, 2007). If a team maintains knowledge assets (illustrated in figure 3), a remote knowledge seeker can already draw contextual knowledge as he is walked through an explanation by a distant team member. On the other hand in absence of any of these assets distance can negatively influence a knowledge sharing process in software organizations. Unlike face to face communications, electronic communication channels cannot make up for the knowledge gaps created by bad software design, inefficient or verbose documentations or other reasons. In their empirical study Ko, DeLine & Venolia found the most common information a developer seeks during code writing activity:

- What data structures or functions can be used to implement this behavior?
- How do I use this data structure of function?
- How can I coordinate this code with this other data structure or function?

(Ko, DeLine, & Venolia, 2007)

These questions, though basic in nature, can be quite time consuming depending on the size of software being worked on. With added complexity of distance a knowledge seeking team member can easily become inefficient; however as mentioned earlier
visual technical illustrations can assist in remote knowledge dissemination by providing cognitive assistance to both the seeker and the guide, as visual knowledge representation is found to be more effective in aiding comprehension (Larkin & Simon, 1987) (Card, Mackinlay, & Shneiderman, 1999).

During my observations in Company-A, it was noted that factors like form of knowledge representation (visual vs text), attributes of software design like standard design patterns and architectural complexity can offset distance disabilities to improve a subject’s comprehension. I carried out a small software design experiment with a member of the UI team in Company-A using hand drawn sketches of a software design instead of descriptive textual form.

Figure 4. Graphical illustration of a system function. Source: Original research
Figure 4 is shows the hand drawn simple illustrations of the functionality. Figure 4(A) is the graphical depiction of currency rounding functional sketch which I used for collaborative comprehension with a developer. The figure facilitated understanding the functionality required and also helped the developer to create simpler implementation. Figure 4(B) illustrates relations between different code assets of Product-A’s feature called price lists. Based on first hand responses from the subject and observation of work done it became apparent that illustrated software functionality was easier and quicker to comprehend for that developer. Agile software development teams that are comprised of non-collocated personnel, can improve their knowledge transfer efficacy by providing visual knowledge assets; as visual information has been noted to facilitate collaborative comprehension. (Laakso, Myller, & Korhonen, 2010); (Myller, Bednarik, Sutinen, & Ben-Ari, 2009); (von Mayrhauser & Vans, 1993).

6.3 Good Design & Learning

Importance of good software design is well understood in the software industry as a means of gaining long term benefits in terms of scalability and extensibility of software. Apart from the technical benefits of good design, during my observations I have noticed and experienced that badly designed software also limits its understandability especially for new team members. A well designed and logically structured software artifact is easy to comprehend as it complies with widely understood standard approaches.

A new individual is more likely to understand standard practice source code and software structure because his/her mental capacity pre-fetches the already familiar concepts underlying the software design. An empirical study points out some developers using API examples to infer purpose of a certain piece of software (Ko, DeLine, & Venolia, 2007). This inference or anticipatory exploration allows team members to self-assist their comprehension of a software system based on their previous knowledge. For instance modular design of software creates clean boundaries and also breaks down complexity into smaller parts allowing better understanding and source code comprehension. Ciliberti refers to this idea highlighting clean software design virtues of Microsoft ASP .NET MVC: “Another advantage of using the MVC pattern is that separating concerns allows different team members to focus on the part of the application that best aligns with their respective skill sets. For example, very few people possess both the skills for creating an attractive front-end interface using HTML and CSS, and also know the intricacies of C# programming” (Ciliberti, 2013).
Good design and standard practices facilitate comprehension by reducing the number of skills and amount of previous knowledge required by an individual to understand a software system. Figure 5 illustrates the potential flow of events leading to knowledge transfer hurdles. On the other hand the same familiarity and existing understanding of a new individual can contribute to confusion if the software under study defies logical design cues and patterns. In a study at Microsoft (Ko, DeLine, & Venolia, 2007) developer inference or hypothesis was also found to be drawn from their intuition and hind sight. These factors that developers use to explore new code are all rooted in their previous knowledge; hence the importance of good software design preferably based on standard industrial patterns and practices.

When we consider the same situation for a new employee, his/her existing knowledge is very likely to be based on software industry norms, standards, patterns and practices. A new employee will therefore base his/her inference/hypothesis based on these norms, standards, patterns and practices. In cases where these aspects within an organization and in the industry have high disparity, knowledge transfer for new employees experience an added layer of complexity, as was the case in Company-A. Figure 1 illustrates the potential effects software design can caste on knowledge transfer procedures.

Observing the software development work process in Company-A, I found the software structure riddled with misused terminologies, which could have been caused by either inadequate language skills in English or technical knowledge. For myself as a new entrant in Company-A, I found terms used misleading and often required investing more time in understanding the problem-solution set and code flow. These factors retard the learning process for a new comer in his/her attempts to understand software; because previous experiences, anticipation of software structure based on logical and industrial standards did not match product-A’s.

Literature on agile methodologies argues that very often these development methodologies result in badly designed software because not much emphasis is placed on to architectural planning and design (McBreen, 2003); (Stephens & Rosenberg, 2003). Company-A portrays this behavior in its product as well. Bad design leads to complex code, difficulties in extending or scaling and hence incur significant overheads. Software systems like product-A, are required to scale for business continuity and bad design acts as a nemesis for scaling (David, Lindvall, & Costa, 2004); so development...
teams circumvent the scaling problems with immediate short sighted fixes eventually leading to even more complexity. Boehm (Boehm, Get ready for agile methods, with care, 2002) points out that agile methods’ reliance on individual tacit knowledge can lead to architectural mistakes which are hard to detect because of lack of documentation; potentially leading to knowledge gaps. A new team member can simply not recognize a mistake and continue investing time in understanding something that was never meant to be. A knowledge seeking subject is more likely to understand logically designed (not necessarily simplistic) software architectures than incoherent impulsive designs. To retain and transfer valuable software knowledge, especially in medium/large software systems, more focus on architectural should be practiced.

Another noteworthy incident is IoC (inversion of control) implementation as described in observations section. The IoC changes were implemented by a subcontractor, team knowledge about this technique was very scarce. Apparently the culture of YAGNI (you are not going to need it) an extreme programming principle, was being followed also in team awareness scenarios. Most of the team members were not interested in exploring and understanding the inner workings of the IoC implementations, nor was this awareness mandated by the team managers. When and if the IoC implementer decides to not continue working in Company-A, this vaguely understood code will incur additional costs of exploring and understanding.

6.4 Technical Illustrations vs Verbose Descriptions

Text based information about a software design, specifications, requirements or functionalities all tend to become quite verbose. Some authors (Ko, DeLine, & Venolia, 2007) point out that developers have lower tendencies going through documents to find an answer. Also my observations in Company-A revealed similar behavior, going through verbose specification documents was not a favored activity. Reading through such documents was tedious and left room for ‘interpretations’ and ambiguities. Observation 9 in Company-A, presented a case of comparative understanding for visual vs textual information for the designated developer. Visual representation of a proposed design was understood quicker and better. Studies in cognitive sciences and computer science have already stated visual that illustration contribute in better collaborative comprehension and reduction of cognitive load (Card, Mackinlay, & Shneiderman, 1999) (German & Hindle, 2006) (Larkin & Simon, 1987). Figure 5, shows how Visual Studio’s dependency graph provides a quick view of system dependencies, which can quickly direct a knowledge seeker to explore the correct source code without having to do time consuming searches.

Moreover when it comes to software in modern days, when architectural complexity of a software system can span multiple processing units distributed all around the globe, several external data end points and other complexities, it becomes even more important to reduce cognitive load of a knowledge seeker. Another layer of complexity in software designs and architectures is added by the unwary adoption of ‘agility’. Agility advocates unplanned (evolutionary) software growth to save time and money otherwise spent on planning. This evolutionary approach, can and very often does lead to more complexity. Some authors have stated their findings in remote collaborative learning with visualizations as more engaging hence more potent (Laakso, Myller, & Korhonen, 2010); (Myller, Bednarik, Sutinen, & Ben-Ari, 2009); (von Mayrhauser & Vans, 1993). Bundling complexity factors with non-collocated teams where intra-team discussion can
be limited to video conference, IM or email a visual document can facilitate knowledge transfer.

A renowned speaker and author in software engineering circles Martin Fowler refers to evolutionary design as usually a disaster. “In its common usage, evolutionary design is a disaster. The design ends up being the aggregation of a bunch of ad-hoc tactical decisions, each of which makes the code harder to alter. In many ways you might argue this is no design, certainly it usually leads to a poor design” (Fowler, Is Design Dead ?, 2014). Harder to maintain code, means harder to comprehend for a developer who is already a part of the team building that software. This difficulty is more pronounced for people who join a software team as new members and have to get up to speed. Having graphical illustrations of a software system not only assists new team members but also acts as quick reference guides naturally for all team members.

Company-A’s business revolves around Product-A, which over the years have accumulated huge source code base and dependencies. Exploring and understanding this system would have been relatively easier if growth of the system was logical and systematic; since development was following an agile scrum method (fairly deemphasized focus on design), structure has become anything but simple. As illustrated in figure 6, one can see how a dependency graph (using Visual Studio dependency graph tool) can provide a concise holistic view of a software system which greatly reduces the amount of time and effort. To comprehend the system, one should understand the bigger holistic picture and structure of the software, reading through
textual descriptions of the system will naturally consume longer time and incur more mental pressure than visualizing a system dependency graph based on high level illustrations. After having an overview of the system, the knowledge seeking team member can further explore and understand by inter or intra-team collaboration. Shneiderman highlights value of visual presentation by: “Overall, the bandwidth of information presentation is potentially higher in visual domain than for media reaching any of the other senses” (Schneiderman, 1996).

Figure 7. Visual Studio CodeMap. Source: MSDN

For my own understanding as a new team member in Company-A, getting into the system was fairly difficult. Fortunately the development tool (Microsoft Visual Studio) provided a very useful feature of generating dependency graphs based on source code as well as code maps. Figure 7, demonstrates how CodeMap provides control flow information which helps in understanding how a piece of code is interacting with other parts of the system. Visualizing a function call through CodeMap creates a high level holistic image of the code’s workings. These two features facilitated my understanding of high level control flows and relationships between different parts of code. I had access to the wiki articles and requirements specifications documents, but reading through those was only overloading my mental capacity. Experienced with designing software I resorted to visually educating myself about the system to attain cognitive, effort and time benefit.

6.5 Team Mavericks & Technical Capacity

Software teams that rely on individuals as the keepers and perpetuators of knowledge expose themselves to risks of knowledge compartmentalization. Extant literature refers to such individuals as maverick programmers (Ravichandran & Rai, 2003). Concentrating software knowledge in individuals creates a dependency that can either make or break a team. If this maverick programmer/designer/architect decides to leave the organization, the organization loses a very valuable knowledge asset. An asset that
is aware of all software evolution milestones, is aware of design decisions and the software architecture.

Agile methodologies specially rely heavily on individual skills and expertise, if individual skills are appropriate a team can accommodate any work process and perform well with it (Cockburn & Highsmith, 2001). The perception of what is appropriate though is a very open ended question here, a very well skilled Pascal programmer might demonstrate very narrow vision of a web based distributed application.

Figure 8. Knowledge implications rooted in technological debt. Source: Original Research.

In figure 8, technical capacity of an agile team is depicted as seeding factor for learning curves. This happens when software development is initiated with shallower technical capacity teams who also want to be purist agile teams. Heeding inappropriate attention to planning and design of a software product eventually incurs learning curves for new members who may be joining the team well after the code footprint is considerably large.

Agile software teams with individual technological debts become very likely to walk the software product into its own technological debt, with or without noticing it. A technology debt can render architecture or software implementation unnecessarily complex hence creating challenges for knowledge sustenance and transfer when new blood is brought on board. New and younger team members might not be able to infer a technique or procedure based on legacy technology vision leading to impeded comprehension.

Observations 2, 5, 8 and 13 from Table 2, portray occurrences in Company-A resulting from original design decisions made by individual designer. Each of these incidents are rooted in historical design and development decisions which were undertaken with little or no consideration for technical flexibility, modern distributed software architecture and industry standards. Observation 13 displays the degree of knowledge compartmentalization in ‘maverick’ based teams, where system extension is performed by an individual and ‘awareness’ levels in the team about this feature are not brought up; therefore creating knowledge obstacles specially for future members. Observation no. 2’s most significant characteristic relates to Cockburn’s argument that good skilled (technical capacity) people can make up for bad procedures and still manage to perform
well (Cockburn & Highsmith, 2001) and vice versa. Martin Fowler too, for this matter says “Any programmer working in high design environments needs to be very skilled” (Fowler, Is Design Dead ?, 2014). In this case a design decision was made that would lead to overhead code instead of writing code within the underlying system constraints. Such approaches create code that circumvent their shortcomings by require additional manipulation of functionality and/or data to achieve a desired result; thus creating two fold problem for knowledge transfer. First by reducing a developer’s inference (anticipatory exploration) and secondly by increasing the amount of code required to understand.

In modern times knowledge management has been identified as an important enough aspect that it merits its own role referred to as team historian or knowledge broker (Morgan, 2010; Ko, DeLine, & Venolia, 2007; D.Latoza, Venolia, & DeLine, 2006). Latoza, Venolia and DeLine attribute historical and architectural knowledge to the team historian. These knowledge aspects naturally can allow an individual to assist knowledge transfer. Designating a maverick as the knowledge broker/team historian role as well, can create very viable knowledge transfer channels. However, the maverick culture also poses a permanent risk of valuable knowledge asset loss.

Extant literature refers to a knowledge broker or team historian as a person who has good degree of awareness (software knowledge & individual’s activities knowledge), can understand the software architecture and can direct a knowledge seeker to the right person or knowledge asset. This person should also be well versed in software architecture and design to be able to pin point immediate knowledge transfer guidelines or answer spontaneous queries. Agile work environments like Company-A, where knowledge assets and team members are not always readily available, a knowledge broker / team historian role can become a very significant improvement.
7. Contributions to Agility

Agile software development teams are prone to fall into the knowledge traps by not focusing on sustaining knowledge and relying very heavily on tacit knowledge. Designs and plans are very important in any kind of software system. Agile methods can be very productive in cases where a highly skilled, polished and experienced set of individuals are put together to form a team, otherwise (and more likely) asymmetric skills and expertise will not produce quality swift outputs.

Software companies, especially software Product based companies should invest more time and effort in creating and sustaining critical technical knowledge assets. As suggested by Boehm (Boehm, 2002), when requirements change less than 1% in a month, opt for traditional software development processes. In case of Company-A which is a product based company, being agile in its purity is not needed as requirements and features decided to be implemented are planned and not likely to change very rapidly. Pure agility in this case takes toll on time given to creating proper knowledge assets that will ensure knowledge sustenance for new members and provide valuable knowledge banks to existing ones. Considering for example observation about implementation of translations (section 5.1) all methods used in the software system for creating translations were not readily visible to new comers, a well drafted knowledge asset can provide quick insights into software knowledge in such cases.

Being agile and putting out a new versions of software with new features is important for business but also very likely to result in haphazard growth of software code. Some may call it evolution, but the term ‘evolution’ cannot be fully applied in such cases. Evolution in nature is a controlled process which always progresses based on attributes of survival; however in an agile team simply by piling up software features after features, logical design starts deteriorating.

In order to ensure immunity to all unseen team turn-over events, an agile software team like Company-A’s team, should modify its development process to include software architecture and its technical illustrations. These documents will continue providing holistic vision of the purpose and technical infrastructure of the system. Also technical illustrations serve as a quick reference of the system design, when extending or modifying the source code. The illustrations should exploit visual knowledge transfer capacity instead of textual forms. Visually understanding a design instead of reading through a long description also improves comprehension of knowledge seeking members of the team. Refer to tools like dependency graphs (http://msdn.microsoft.com/en-us/library/dd409453.aspx) for an example of conciseness and potency of visual depiction of software architecture.

Distributed agile teams as well, can draw benefits from such knowledge assets as they greatly facilitate assisted comprehension for new team members. Product based software companies, can continue to be agile in terms of implementation of a software feature without attempting to be agile in planning and design of software. Design and plan should be carried out with care as it will not only preserve valuable knowledge for future team members but eventually dictate scalability, extensibility and maintenance of the software product.
Based on literature analysis and observations carried out in Company-A regarding their development process, following illustration outlines a work flow that creates and disseminates knowledge to new members by providing them holistic views of the system.

Knowledge sustenance in a software organization spans the entire life cycle of a software project or product. Looking at the knowledge obstacles (described in section 2) one sees that creation of knowledge can be negatively affected by these. On the other hand if an organization follows industrial standards, reduces technological debt, and creates and facilitates visibility it can knowledge sustenance. As mentioned earlier knowledge sustenance depends on whole life cycle of software, starting from how software is designed and developed to ensure the standards compliance and reducing technological debt is a step that increases the degree inference. Increasing the inference capacity of a new team member improves learning and can shorten the time required for an individual to become productive in terms of understanding the software artefacts. Emphasizing design and planning of a software artefact also creates knowledge assets that sustain valuable knowledge for team members and new entrants as well. Creating the knowledge assets in textual form is seen as another obstacle because of its verbose nature and high cognitive load which directly relates to the complexity of software. Therefore creating knowledge assets in visual forms with illustrated documents facilitates understanding and decreases cognitive load. Software design and planning can ensure knowledge sustenance especially for agile software teams and also facilitate them to be agile by allowing efficient knowledge transfer to new team members. The utility of software design and special skills needs of agile team members are stated by Martin Fowler (Folwer, 2014) and Deursen (Van Deursen, 2001). It’s very unlikely that a team of software developers is entirely composed of above average developers, and hence knowledge transfer becomes important not only for new members but also for existing teams to create visibility and awareness.

Also by including more focus on software design/architecture a software team can create technical knowledge assets that will ease cognitive load. Teams like Company-A,
who work on product lines where each release is planned and features already set, more focus on design will ensure better long term scalability of the software system. Spending more time on design may appear as an action against agility, but in software environments where requirements volatility is low, including features of traditional plan based software development models can ensure positive effects like design conformance, scalability and extensibility.

Agile teams, especially scrum based ones have release plans where content of release is chosen which subsequently determine the direction of development endeavors during every ‘sprint’. Figure 9, suggests adding knowledge transfer features to the release plan and adds a phase for design and architecture of the software based on the release plan. Both these phases produce knowledge assets. Agile team can then use these knowledge assets for collaborative comprehension, remote or onsite. One enabling factor for any programmer to start working on a software system is the degree of understanding s/he has of the system. A software system’s purpose (functionality) and infrastructure (architecture) are of immediate importance. These aspects facilitate further understanding and exploration. Replication of real life functions for instance tax calculating involves a data set being operated upon by a function that produces a result. In this example this function is a tax calculation formula. Experiences from Company-A, indicate comprehensibility of textual descriptions of a software’s functions starts deteriorating with its size and complexity. However, visual representations of a function i.e. a tax calculation can provide clearer understanding of data relations, functions applied and other relationships or dependencies. Similarly technical understanding of the structure and architecture of the system precedes any code writing capability of a new team member. Once a member is assimilated into the knowledge domain, s/he is mentally better positioned to infer, explore and understand the software.

Deeper and more detailed knowledge transfers of actual source code and problem spaces can be transferred with relative ease in a collaborative manner. Agility and distributed teams have their implications on knowledge creation and sharing as discussed earlier in this thesis, but they can be offset by providing knowledge assets to new members which do not incur high cognitive load (i.e. visual knowledge), better design of software and standard patterns and practices which have wider familiarity and understandability in the software industry.

Software development especially in agile environments requires truly skilled teams. If a software organization does not invest enough time and effort in recording and sustaining the knowledge thus created i.e. externalizing knowledge, most of the knowledge stays in tacit forms held by individuals. In such organizations the importance of co-location becomes paramount, since co-location facilitates tacit knowledge sharing (Chau, Maurer, & Grigori, 2003), besides mandating co-location tacit knowledge also poses challenges to knowledge sharing when formal methods of knowledge sharing have to be used for example in case of remote members or sub-contractors. (Chau & Maurer, 2004). Boehm (Boehm, 2002) points out that if teams are capable of utilizing their tacit knowledge assets things work fine. Though he also mentions that not all software organizations can draw equal advantage from agile methods. For example observation of KPI implementation (section 5.1) when the knowledge holding individual is located elsewhere, software design documents with visual illustrations can facilitate knowledge transfer by reducing cognitive load and providing contextual information about how each software artefact relates to the rest of the system.

Agile methods like pair programming that claim to rectify knowledge sharing by suggesting pairs of expert and novice developers, fall short because of constraints posed
by human social behavior (Chau & Maurer, 2004). This essentially means team
dynamics play important role in co-ordination, work practices and hence knowledge
creation and sharing. My observations in Company-A (no. 14) have noted that some
members very eager to create and share knowledge and some isolating their
responsibilities from others creating awareness/knowledge gaps.

Software businesses working on product lines like Company-A, usually do not have to
deal with rapidly changing requirements as all features of the product are planned
within the organization for every release; so opting for a quick to market and rapid
adaptability method is not necessary. My observations (3, 4, 6, 9, and 10) provide
references to the fact that agility (as seen by purists) rendered loss of knowledge and
mandated additional co-ordination with remote team members. Additional costs of
knowledge were also incurred because knowledge seekers faced absence of contextual
knowledge or documents to facilitate assisted comprehension of the software.
8. Limitations of Study & Further Research

This study has drawn upon prior research done in software engineering, cognitive sciences and information processing in addition to the field observations carried out as part of the research work for this thesis. Though the findings and inferences made in this study are based on prior research as well as the empirical cues, the duration of data gathering was only based on one team and a relatively short period of time (1 year).

Software development is a knowledge derived profession and relies heavily on people as keepers and propagators of the related knowledge, hence the transfer of knowledge also depends social behaviors of a team. This study does not analyze effects of human behavior, organizational politics and other related areas on knowledge creation and sustenance.

Although based on literature value of graphical knowledge representation has been found to be superior to textual representation only, this study does not further elaborate on what exact kinds of technical illustrations provide the most value for comprehension. Diagraming techniques like UML or other system design methodologies all have their pros and cons. Although UML provides a set of different diagrams which each serve a different purpose by portraying different aspects of a software system.

Similarly the study has only collected and drawn upon data from a team (Company-A) who claims to be agile, and hence the data about knowledge creation and gathering cannot necessarily apply on teams that work via traditional software engineering / development models.

Further research in the area of human behavior both on individual and social levels, will shed more light and distill more concrete factors that interact and influence other factors described in this study. Human / social behaviors should be studied in the light of this study for their effect on knowledge creation and sharing in software organizations. On the other hand, when considering software development models, there is room for further study in exact applications of agile methodologies in organizations to distinguish if knowledge problems arise from incorrect application of agile methodologies or is it simply the principles of agile methodologies that weaken a team’s perception of knowledge creation and importance of sharing the knowledge.

Another aspect that was out of scope is, social aspects of agile software teams and how they can effect work practices and eventually influence knowledge creation and sharing. A team’s collective behavior dictates its efficiency and how well it can manage its knowledge. True agility is more than a collection of practices; it’s a frame of mind (David, Lindvall, & Costa, 2004), this applies also to the knowledge creation aspects of agility, as long as people and teams’ vision of agility relies on purist approaches of ‘right now right here’ explicit knowledge creation will suffer.

Conducting participatory research as a participant observer might render the researchers views subjective. Therefore the work presented in this thesis might also portray attributes of subjective studies. Observations made were made from the point of view of a software designer in a team, and might overlook or differ when carried out from
another perspective i.e. team manager or scrum master. Further research can also be carried out by observing suggested changes in an agile team for a longer period of time.
9. Conclusion

Agile methods are not a ‘one size fit all’ solution. Software business environments that house well skilled teams, have low employee turnover and high requirements volatility have no choice but to be agile. In agile teams knowledge is held by individuals, who are after all humans and are likely to either switch jobs, move away or just not work in the same organizations forever. When and if they leave, they take the knowledge along, since agile methods focus more on ‘acting’ then planning and documenting. Agile software methods rely on highly skilled team members. When teams change specially in agile organizations, software knowledge is at risk of loss with every departing employee. On the other hand when a new employee comes in, transferring knowledge and quickly bringing him/her up to speed is critical; the team is agile. Quick response is the key in being agile but the paradox of comprehending an undocumented software creates hurdles in induction processes of new employees. As the research question of this thesis was:

*What are the factors influencing efficiency of software knowledge transfer to new members in agile teams?*

The influencing factors in terms of software development processes are identified through literature and observations. There are socio-technical and cultural aspects influencing knowledge transfer as well but they are considered out of scope for this thesis. The factors identified are firstly the obstacles that directly hinder knowledge transfer i.e. non-compliance with standards and technological debt; these two factors effect inference capacity of an individual eventually influencing the efficiency of knowledge transfer. Then there are knowledge visibility and visualization that refer to knowledge representation forms; visual forms of knowledge (diagrams, charts and illustrations) can increase efficiency of knowledge transfer.

Literature analysis and the observations conclude that merely creating manuals and verbose documentations of code do not contribute much in transferring knowledge to new employees; long descriptive texts can cause cognitive overload and dissuade a knowledge seeker from exploring all together. Hands on face to face conversations are very effective method, but they are unfortunately not always available as software teams can be distributed, team members could be travelling etc. and distributed teams cannot fully exploit the spontaneous knowledge seeking hence mandating formal knowledge assets for the purpose. As illustrated in Figure 9, knowledge creation steps in an agile workflow can ensure continued knowledge availability for new members and also facilitate collaboration/training with both onsite or offsite team members.

New team members come in with their existing knowledge of software industry, which is very likely to be comprised of standard approaches, design patterns and practices which are taught and utilized across academia as well. New team members can utilize their knowledge to explore and learn the software system in their new team. But if the system under question is haphazardly designed, or built with legacy or proprietary patterns not widely known in the industry a new team member may not be able to use his/her existing knowledge to understand this system. A software system with logical structures and control flow allows a new comer to apply his/her known concepts to the learning process.
Most medium to large scale software development endeavors take a significant amount of time, money, effort and collective skills and knowledge of individuals. Software industry solely relies on a knowledge held by individuals (Boehm, 2006). To ensure continuous effective operations of a software organization knowledge plays a very important role. If developers cannot easily understand the inner workings, domain and context of software, they cannot perform quality software development and incur additional costs for the organization.

Creation and sustenance of software knowledge hence is critical for software organizations. Sustenance involves preserving the knowledge held by leaving employees and effectively and efficiently transferring it to new employees when needed. Knowledge of software design and the inter-relations between different artefacts is very important in making a new member productive. As software systems grow in complexity, grasping this knowledge can become daunting, hence the need of a low cognitive load approach which not only provides the immediately needed software knowledge but also presents the relationships with other parts of a software system. e.g. CodeMap visualizer, also (Card, Mackinlay, & Shneiderman, 1999) state the benefits of visualizations in improving cognition and hence for knowledge transfer.

Agile philosophy holds adaptability and ability to quickly respond to environmental (pertinent to software organization) changes, at its core. If a software process results in knowledge sharing obstacles, perhaps it would be more agile to adapt the process and accommodate for knowledge sustenance.

In order to successfully create and sustain knowledge, extensive measures to handle social aspects of a software development team (team dynamics) should be taken. Team dynamics directly influence work practices and hence can make or break a knowledge and awareness creating culture. In agile teams, selection of team members is very vital, though agility should not dictate complete dismissal of technical knowledge recording i.e. technical illustrations. Knowledge transfer for new employees especially can be assisted through visual knowledge representation by providing them quick references to a holistic picture of software, domain logic and contextual information. Further assistance from older team members can provide a practical walk through to provide even more facilitation.

Technology debt and low technological capacity of an organization can influence how easily its knowledge can be transferred to a new member by decreasing inference, during my observations I also noticed periodic unhappiness in some developers about being stuck in older complicated techniques. With passage of time every new member will have higher probability of not being familiar without date software practices. Students fresh from a training school or another, usually learn standard industrial patterns and practices, in such cases the learning curve they would have to go through to get up to speed with a legacy code base will but much steeper. Proper technical skills set, visual knowledge records, standards and work practices all contribute to the efficacy of a software organizations knowledge transfer capacity. Continuous effective knowledge transfer procedures should be in place to ensure positive growth of a software organization (Baron, 1996) and that requires that all contributing factors be balanced and tweaked.

Exploring huge sets of information can be associated with anxiety or cognitive overload (Wurman, 1989), with software systems become huge collections of source code and other related information with added learning curves of design complexity, tech debt and remote information sources (individuals) it becomes even more important to create
quicker and easier to comprehend knowledge assets. Extant literature and observations together provide support for the notion that Knowledge transfer in agile teams is influenced by the development process, software architecture, team structure and the form in which knowledge is externalized; hence can be improved by bringing improvements to these factors.
References


Appendix.

Definition of Terms

**Anticipatory Exploration** refers to the phenomenon of a software developer inferring the flow of control in source code based on his or her previous knowledge of software designs and implementation details i.e. patterns and practices. A very simplistic example of this concept is knowing JavaScript ‘click’ function are run when a UI button is pressed; therefore when a developer is required to change click behavior of a button, s/he would naturally start looking for a function bound to that button. This pre-known pattern hence provided him with some contextual information, facilitating his comprehension of the code in question.

I have used ‘Anticipatory Exploration’ as a term to refer to a class of comprehension, where a human subject can anticipate the functionality of software source code, snippet, module or system by looking at high level organization/structures. This speeds up further exploration and comprehension because the subject starts exploring in a logically coherent direction and pace.

**Behavior** refers to the execution of the program with real and abstract data. The execution can be seen as a sequence of program states, where a program state contains both the current code and the data of the program. Depending on the programming language, the execution can be viewed on a higher level of abstraction as functions calling other functions, or objects communicating with other objects.

**Evolution** refers to the development process of the software system and, in particular, emphasizes the fact that program code is changed over time to extend the functionality of the system or simply to remove bugs (Diehl, 2007).

**Efficiency / Efficacy / Usefulness** when referring to software knowledge assets the term efficiency describes the capacity of a knowledge asset to transfer the knowledge represented to a knowledge seeker with minimal ambiguity. When applied to software development or developer these terms refer to how quickly the individual can absorb knowledge represented by the software knowledge assets and contribute to the software tasks. Efficiency, efficacy and usefulness do not appear in any order in the text neither do they have preferential meanings based on the order they appear in this definition. In context of software developers and knowledge assets these terms may be used interchangeably in this document.

**Structure** refers to the static parts and relations of the system, i.e. those which can be computed or inferred without running the program. This includes the program code and data structures, the static call graph, and the organization of the program into modules.