ABSTRACT

The goal of project management is the timely delivery of capabilities that provides value-add to the customer. This paper identifies a series of warning signs of Information Technology (IT) projects under stress and their potential effects on technical quality. It describes how agile software development practices can reduce the causes of these stresses. It then examines the Manifesto for Agile Software Development and how it’s implementation in six key areas could also result in a reduction in technical quality and provides recommendations for Project Managers to follow to maintain technical quality while executing agile software development practices.

Key Words: Manifesto for Agile Software Development, Technical Debt, Tailoring, Customer Definition, Architecture Communications, Documentation, Self Organizing Teams

This paper is based on empirical observations, current literature, limited trails, and engineering and project management experiences.

INTRODUCTION

The goal of project management is the timely delivery of capabilities that provides value-add to the customer. In 1994, the Standish Group published their first report that measured the delivery of successful projects. In their initial report, they surveyed 365 companies and represented 8380 IT applications. “The Standish Group research shows a staggering 30.1% of projects will be cancelled before they are completed….On the success side, the average is only 16.2% for software projects that are completed on-time and on budget.” They further assessed that 52.7% of IT projects will be completed but will be challenged by being over-budget, over the time estimate and will offer fewer functions than originally specified. The study also concluded that the probability of project success goes down as the size and complexity of the project increases. (The CHAOS report.1994)

As reported by Alan Zucker in 2016,

“Over the past two decades, there has been very little change in the (Standish) headline results. On average:

- 29% of projects “succeed” in delivering the desired functionality, on time and on budget
- 48% of projects are “challenged” and do not meet scope, time or budget expectations
- 23% of projects “fail” and are cancelled
While there is some year-to-year variability in these scores, the trend line is essentially flat. In other words, we are no better at delivering a project today than we were 20 years ago. However, when you dive into the data, there are some bright spots and markers for improvement:

- Smaller is better
- People are the primary drivers of project success or failure
- Agile projects are far more likely to succeed.”  (Zucker, 2016)

Empirical observations by the author of approximately 150 projects over a period of 20 years indicate that troubled projects proceed through a progressive set of warning signs until the project either delivers a capability or is cancelled.

![Progressive Warning Signs of Project Stress](image)

**Figure 1: Progressive Warning Signs of Project Stress**

The first warning sign of a project under stress is typically seen in planned delivery dates for task completions being missed, or an output is delivered on the scheduled completion date but it is of very poor quality. Figure 1 provides a set of potential causes for schedule dates being missed and nearly all of them relate to worker productivity. Most likely, the reason why missed dates is the first warning sign is because schedule, as a resource, is consumed irrespective of any working being accomplished.

The next warning sign of a project under stress is typically seen as cost overruns. To return the project to its planned schedule means that the existing workforce has to consume more labor hours, or additional personnel resources are added to the project, which again consumes more labor hours. The additional labor hours required for the project may translate into an additional cost for labor for the project. It should also be highlighted that both schedule and cost are two project management parameters that are very easy to collect, objectively measure and report. Thus, these measures provide easy to produce warning signs.
With the project now expanding beyond its bounds of schedule and cost, the last warning sign is to reduce the technical scope of the project to ensure that the most important, value-added features are delivered. But even with a reduction of technical scope, if there are still schedule and cost pressures, then the technical quality of the delivered capability is still reduced. Unfortunately, technical quality is a very difficult parameter to collect and objectively measure.

![Diagram](figure2.png)

**Figure 2: Differences in Traditional Waterfall vs. Agile Cost Variables** (Modigliani & Chang, 2014)

Figure 2 provides a diagram that conceptually identifies the major differences between traditional, plan-driven development methodologies, such as waterfall, and agile development methodologies:

“As illustrated….under traditional waterfall development projects, scope and quality are fixed, while schedule and cost vary and generally increase as requirements are further defined during the project. Agile development projects attempt to fix schedule, cost, and quality while adjusting the scope of specific releases to fit these constraints. The scope of each delivered release is defined by the prioritization of requirements that deliver end-user functionality within the constraints of the cost, schedule and quality parameters. As a result, treating cost as an independent variable drives the prioritization of requirements and facilitates release planning.” (Modigliani & Chang, 2014)

In both methodologies, technical quality is assumed to be a fixed entity.

**THE MANIFESTO FOR AGILE SOFTWARE DEVELOPMENT**

Plan-driven methodologies approach a project’s development methodology from a risk-adverse perspective. As Alistair Cockburn writes:

“We have developed, over the years, an assumption that a heavier methodology with closer tracking and more artifacts, will somehow be “safer” for the project than a lighter methodology….The heavier-is-safer assumption probably comes from the fear that Project Managers fear when they can’t look at the code and detect the state of the project with their own eyes. Fear grows with distance from
the code. So they quite naturally request more reports summarizing various states of affairs and more coordination points.”  (Cockburn, 2007)

Barry Boehm and Richard Turner voice similar concerns:

“Plan-driven methods have had a tradition of developing all-inclusive methods that can be tailored down to fit a particular situation. Experts can do this, but nonexperts tend to play it safe and use the whole thing, often at considerable unnecessary expense.”  (Boehm & Turner, 2004)

In February 2001, a group of seventeen highly knowledgeable, experienced, expert software practitioners met at The Lodge at Snowbird ski resort in the Wasatch mountains of Utah to discuss an alternative to documentation driven, heavyweight software development processes. The result was a Manifesto for Agile Software Development which defined four value statements and documented twelve principles.  (Manifesto for agile software development, 2001)

Multiple software development methodologies, such as Extreme Programming (XP), Scrum, Dynamic Systems Development Method (DSDM), Adaptive Development, Crystal, Feature-Driven Development (FDD), and Test-Driven Development (TDD), implement these value statements and principles in different ways, but these software development methodologies provide many advantages to a Project Manager and addresses many of the causes of project failure listed in Figure 1.

For example, all agile methodologies implement a prioritization of requirements mechanism that focuses on the early delivery of the highest value requirements first.  As Larry Apke writes:

“The phrase ‘valuable software’ reminds us to always be vigilant that we are actually concentrating our efforts on the most valuable stories, those that will give the most return on investment.  We should keep in mind the Pareto principle—that we receive about 80% of our benefit from 20% of our stories and that means there is a huge value in the work not done….In 2002, Jim Johnson of the Standish Group….presented findings of feature and functions used in a typical system. The number of features that were never or rarely used totaled a whopping 64% while sometimes, often and always weighed in with 16%, 13% and 7% respectively.”  (Apke, 2015)

These methodologies also have mechanisms that reduce and allocates the level of commitment for the delivery of capabilities to the customer.  For example, the project may only commit to the features in a release that we will be delivered in three to six months.  This commitment is then allocated to the features that will be delivered within each iteration of a release.  The development of these features is then tracked on a daily basis by the team.  From an internal perspective and on a daily basis, each team member is committing to delivering something each day and because of this, it is extremely difficult to pull a team member away to perform another task.

Daily standup meetings improve accountability and reduce procrastination, while also quickly identifying and resolving impediments.  Troy Dimes describes daily standup meetings and common challenges in agile methodologies such as Scrum:
“Every team member gets to talk about what they’ve already accomplished, what they’re planning for the day, and what their impediments are….In the Daily Scrum, the team members aren’t really reporting to the Scrum master, but to each other….If a team member consistently fails to do his task on time, it’ll reflect on the backlogs. Whether it’s due to poor time estimation or even lack of skill, Scrum will eventually bring that into surface, allowing the team to take the necessary steps to ensure that the next sprint happens more smoothly than the last. Scrum makes every team member more responsible since a lot of problems they have would most likely be reflected visually in charts and backlogs. While this may discourage some team members at first, in the long run it helps the team improve as a whole and makes the members more accountable in their own decisions. It teaches team members to be relatively independent.” (Dimes, 2014)

Agile methodologies have mechanisms for reporting software development progress. Some methodologies use a visual board which tracks each feature or user story through the iteration to completion. The allows the entire team so visually see progress and determine whether they’re about to be late in delivering the product. Similarly, burndown charts show how much work is left each day until the iteration ends. More importantly however, is that each method includes an activity whereby the team has to report the completion of an iteration with an actual demonstration of the working software, a truer measure of progress. Unfortunately, most demonstrations show that features have been developed but rarely gives insight into the actual quality of the software for achieving performance under loads, or the ability of the software to be maintained over the life time of the system.

TECHNICAL DEBT

Although agile methods can reduce the potential causes of project failures, it should be highlighted that agile methods can also introduce a new category of problems; the most significant is the accumulation of technical debt.

“Software projects often cut corners in the rush to meet deadlines, resulting in bad code….Most of us have experienced occasions where we’ve been required to take short-cuts to make delivery deadlines….Taking short-cuts generally means that the next time the software is touched, it needs to be fixed before any further work can be done. So the second piece of work is even more expensive to do correctly than the original piece of work, and you can be sure that the deadlines are just as tight as they were the first time. Worse, developers generally prefer to play it safe—if someone has left them a dodgy-looking piece of code, they prefer to leave well enough alone….So, unless there are strong individuals present who are really dedicated to good engineering, the team takes another short-cut and works around the code affected by the previous short-cut. The third change invokes working around the first two short-cuts and so on. If one follows the trend to its logical conclusion, and in my experience many teams do, one finds the code complexity grows at an increasing rate. After several changes to the software, it reaches the point where nothing can be changed without significant time and risk. Usually at some point, the team begins to realize that they need to fix things they’ve broken. But by then, it’s too late because they are spending all their time
just keeping the fragile system running and have no spare capacity to fix the code. They’ve painted themselves into a classic Catch 22 situation.” (Brazier, 2007)

Ward Cunningham coined the term technical debt as a metaphor for the trade-off between writing clean code at higher cost and delaying the delivery and writing messy code cheap and fast at the cost of higher cost of maintenance efforts once it’s shipped (Cunningham, 1992). Frank Buschmann states “Technical debt is similar to financial debt; It supports quick development at the cost of compound interest to be paid later. The longer we wait to garden our design and code, the larger the amount of interest.” He then describes various strategies for managing technical debt which includes:

- Debt repayment – initiating a large re-engineering effort to clean up the mess
- Debt conversion – executing an effort that resolves near-term performance problems and reduces maintenance cost, but does not fully retire the debt, similar to refinancing a financial debt to obtain a lower interest rate
- Interest payment – continue to incur the debt and pay the interest through higher maintenance costs
- Debt retirement – Eliminating of the debt by retiring the system (Buschmann, 2011c).

Buschmann discusses three methodologies for gardening systems; refactoring, reengineering and rewriting. Refactoring improves the development quality of a part of system while preserving its functional behavior. It may address the modularization of common code, enhancing component interfaces, or adjusting the flow of code to optimize performance. Refactoring is typically focused on small structural improvements limited to single system elements (Buschmann, 2011a). He describes reengineering as:

“a systematic activity to evolve the software to exhibit new behavior, features, and operational quality….Reengineering alters the design and realization of software through a series of system-level disassembly and reassembly activities. Its goal can be to:
- improve a system’s structural quality
- boost its operational qualities or
- provide entirely new functionality” (Buschmann, 2011b)

Lastly, Buschmann states that “if reengineering it too costly, you could consider rewriting. This means replacing an existing system or component with an entirely new implementation….but) it might be necessary to keep the old system alive while the new one is under development.” (Buschmann, 2011b)

The Manifesto of Agile Software Development defines values and principles but does not offer any specific implementation guidance. Applying the values and principles to the specific needs and risks of a project is appropriately left for the Project Manager and team to address. If taken literally and without thought, the execution of the manifesto can introduce technical debt. The remainder of this paper discusses six topics where a Project Manager should focus his team as they initially draft an agile methodology for their project. They are:

- Defining done
• Defining the customer
• Architecture
• Communications
• Documentation
• People

Figure 3 maps these six discussion topics to the affected values and principles in the manifesto.

Figure 3: Mapping Discussion Topics to the Manifesto for Agile Software Development

**DEFINING DONE**

As previously stated, an agile team should concentrate their efforts on the most valuable stories, those that give the most return on investment and that 80% of the benefit comes from 20% of our stories, consequently there is a huge value in the work not done (Apke, 2015). Unfortunately, the principle of “Simplicity – the art of maximizing the work not done—is essential” is frequently interpreted as not needing to perform many routine activities, such as software version control, that ensure the integrity of the system throughout its life cycle. Figure 4 examines the concept of tailoring. The left diagram shows the increasing cost of executing all components of a risk-adverse, plan-driven development methodology. It is generally easy to quantify these costs and consequently they are generally well-known to the project’s stakeholders and participants. The middle diagram portrays the risk of not performing one or more components of a methodology. The chart conceptually translates risk into a cost value should the risk be realized. For example, an agile team delivers a system that contains 100 software modules, but
then in subsequent releases the team only baselines the subset of modules that changed during the release, thus the work and cost of deploying each individual release is small. Left attended, this deployment methodology could occur many times over many years. But then a disaster occurs in the data center and the system has to be completely rebuilt. This would require the team to go back to the initial release and then re-apply all subsequent releases to bring the system up-to-date. Not only is more time required to rebuild the system, but the organization experiences a longer outage time, which could affect near-term revenue streams as well as long-term organizational reputation. The cost of versioning a complete build release is well known. The risks and costs of not doing something are not easy to calculate but could have a tremendous impact on an organization if realized.

![Figure 4: Tailoring Requires Balance Between Risk and Process (Walden, Roedler, Forsberg, Hamelin, & Shortwell, 2015)](image)

The right diagram in Figure 4 identifies a mid-point between being “blind” to these risks and being “adverse” to these risks. This mid-point is defined through a process called tailoring and the mid-point is different for each project.

“Factors that influence tailoring at the project level include:

- Stakeholder and customers (e.g., number of stakeholders, quality of working relationship, etc.)
- Project budget, schedule, and requirements
- Risk tolerance
- Complexity and precedence of the system

…..Common traps in the tailoring process include, but are not limited to, the following:

1. Reuse of a tailored baseline from another system without repeating the tailoring process
2. Using all processes and activities “just to be safe”
3. Using a pre-established tailored baseline
4. Failure to include relevant stakeholders.” (Walden et al., 2015)
By executing the tailoring process, it should accelerate the team’s agreement and understanding on the definition of done.

“As a recap, potentially shippable products are products that are completed from sprint cycles. In order to be truly completed though, everyone must agree on standard conditions on what completed work really means. One member may view completed work as something that just works whereas another member may view completed work as something that works, has proper documentation, and thorough testing. If team members don’t have a unified idea on what completed tasks should be, the performance of the whole team will inevitably decline….Having a unified definition of completed work is especially important if multiple Scrum teams are working on one project….The idea on what’s completed and what isn’t is a hard one to standardize. People on the mediocre side tend to be more complacent, whereas people in the perfectionist side tend to be more strict and rigid. Their ideas on what ‘completed’ means will not always coincide, which is why the team needs to work together to have a unified definition of the concept. A unified idea of completion is not something that’s achieved overnight, but as teams work on more projects together, soon enough, they’ll have a better grasp on what completed tasks should really look like.” (Dimes, 2014)

The definition of done can be different for a completed feature/story, for a completed iteration/sprint, and for a release into production.

“For a user story, the definition may include code completion, the level and types of testing, and (just enough) documentation. For a release, the definition may include more rigorous testing, such as regression testing, certification, product owner approval and release build.” (Modigliani & Chang, 2014)

DEFINING THE CUSTOMER

The manifesto includes one value statement and four principles that either specifically use the word “customer” or use similar terms such as “business people” and “sponsor.” The manifesto infers that the overall goal is the continuous delivery of valuable software for the customer’s competitive advantage. To appropriately define customers, then there needs to be some definition of who will either create or obtain value from the software. Table 1 provides an overview of the many entities that comprise this value stream.

The agile methodologies typically assign a person to represent the needs of the customer. The person, called the Product Owner is Scrum, manages the product backlog and defines and accepts completed features or user stories. In many situations, the Product Owner typically represents the customer and operators of the system and they are focused on the delivery of new functions or features of the system. They may have insight into how the timely delivery of specific functions and features will provide a competitive advantage in the market place. Unfortunately, very few Product Owners have the technical expertise to ensure that non-functional requirements, such as performance under-load, availability, disaster recovery, and
information security are addressed to ensure business continuity and customer reputation are maintained in the marketplace.

**Table 1: Entities That Create or Obtain Value from the Software**

<table>
<thead>
<tr>
<th>Type of Customer</th>
<th>Position in the Value Chain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shareholders, Owners, Corporate Executives, Sponsors</td>
<td>Revenue and profit generation through reputation, business continuity and competitive position in the marketplace</td>
</tr>
<tr>
<td>Customers of the system</td>
<td>Willing to pay money to obtain the services provided by the company and its systems (e.g., travelers on an airline)</td>
</tr>
<tr>
<td>Operators of the system</td>
<td>Use the system to deliver value to paying customers (e.g., airline travel agents, gate clerks, baggage handlers, customer service representatives)</td>
</tr>
<tr>
<td>Maintainers of the system</td>
<td>Keep the system up and running such that value can be continuously delivered (computer support technicians)</td>
</tr>
<tr>
<td>Developers of the system</td>
<td>Develop new and updated capabilities that provide more value to the organization (future developers)</td>
</tr>
</tbody>
</table>

System maintainers are focused on functions and features that either prevent a system outage or minimize the duration of an outage. They are also interested in capabilities that minimize the manual effort required to keep a system operating at peak efficiency. As application software becomes more complex and hardware setup and operating system provisioning, especially in a cloud environment, are now being executed through software provisioning scripts, system maintainers are now becoming members of agile teams through a practice called DevOps.

Lastly, future developers of the system need to be considered in this definition of the customer. Agile methodologies embrace the notion that the system will be continuously changing over its life cycle and thus any system decisions must embrace the ability to make change. As Dave Thomas, one of the authors of the manifesto, states “A good design is easier to change in the future than a bad design….When faced with two more alternatives that deliver roughly the same value, take the path that makes future change easier.” (Thomas, 2015)

**ARCHITECTURE**

The manifesto has one value statement and two principles that affect architecture. One states that “the best architectures, requirements, and designs emerge from self-organizing teams” while the other two state that the team should welcome and respond to change for the customer’s competitive advantage. Implementing these elements without applying energy to architecture can have serious consequences. As Roger Sessions states:

“Architectures naturally seek the maximum possible level of complexity all on their own. If it is a complex architecture you are after, you don’t need architects. You might as well just fire them all and let developers work on their own. This observation that architectures are naturally attracted to complexity is actually predicted by physics—in particular, the law of entropy. This fundamental law of physics states that left to their own, all systems evolve into a state of maximal disorder (entropy). It takes a constant inflow of energy into a system to keep the disorder at bay. In this regard, enterprise architectures are just another natural
system, like gas molecules in a balloon. The law of entropy tells us that the battle for simplicity is never over. It requires a constant influx of energy to keep enterprise systems simple. It isn’t enough to design them so that they are simple. It isn’t enough to even build them so that they are simple. You must continue working to prevent an erosion of simplicity for the life of the system. In this sense, the work of the enterprise architect is never done.” (Sessions, 2008)

Successfully incorporating architecture into agile projects is a difficult balancing act.

“Companies where architectural practices are well developed often tend to see agile practices as amateurish, unproven, and limited to very small, Web-based sociotechnical systems. Conversely, proponents of agile approaches usually see little value for a system’s customers in the upfront design and evaluation of architecture. They perceive architecture as something from that past, equating it with big design up-front (BDUF)—a bad thing—leading to massive documentation and implementation of YAGNI (you ain’t gonna need it) features. They believe that architectural design has little value, that a metaphor should suffice in most cases, and that the architecture should emerge gradually sprint after sprint, as a result of successive small refactoring….The tension seems to lie on the axis of adaptation versus anticipation. Agile methods want to be resolutely adaptive: deciding at the ‘last responsible moment’ or when changes occur. Agile methods perceive software architecture as pushing too hard on the anticipation side: planning too much in advance.” (Abrahamsson, Babar, & Kruchten, 2010)

Grady Booch provides insight into the birth and maturing of many systems and the need for architecture:

There are many examples of notable systems that began with the code of one or two people and grew to become a dominant design: the packet-switched multiple-protocol router, first developed by Bill Yeager; a graphics editing system, first developed by Thomas and John Knoll; a social network, first popularized by Mark Zuckerberg. The list goes on. In each of these cases, architecture was not a primary concern. I’d be surprised if it was on their radar at all, save for the reality that each of these developers had the chops, the experience, and the intuition to deliver something Good Enough that could be grown….Quite often, the developers who did the internal exploration are not the most skilled at production. Furthermore, the risk profile changes, and the success of a system is less dependent on rapid innovation and much more dependent on quality and efficiency in manufacturing and delivery….it’s also these times that intentional architecting becomes intensely important.” (Booch, 2011)

The following discussion provides insight into the application architecture processes into an agile activity:

“Do not dilute the meaning of the term architecture by applying it to everything in sight. Not all design decisions are architectural….apply architecture) early enough because architecture encompasses the set of significant decisions about
the structure and behavior of the system. These decisions prove to be the hardest to undo, change, and refactor, which means to not only focus on architecture, but also interleave architectural ‘stories’ and functional ‘stories’ in early iterations….User stories in agile development relate primarily to functional requirements; this means that nonfunctional requirements can sometimes be completely ignored. Unfulfilled nonfunctional requirements can make an otherwise functioning system useless or risky. A main objective of integrating architectural approaches in agile processes is to enable software development teams to pay attention to both functional and nonfunctional requirements.” (Abrahamsson et al., 2010)

Richardo Valerdi identifies five reasons for why architectures matter:

“First, they enable designers to document assumptions and understand the structure of their creation….Architects use architectural patterns that leverage reusable solutions to commonly recurring problems….Second, architectures provide the ability to establish design baselines and perform trade studies, also known as ‘what-if’ analyses….Third architectures help clarify details at the micro and macro level. End users might be interested in the detailed design as it affects efficiency, security and implementation. Owners might be interested in business requirements and environmental contexts of a system that may change over time, such as legal, social, financial, competitive, and technology concerns. Fourth, architecture allows reuse of components between projects….Fifth, architectures provide insight into the maintainability of a product. The architecture that is originally implemented might change over time as improvements are introduced. This change might cause an architectural drift between the planned and actual architecture of the product as realized in his implementation….The gap between the planned an actual architectures is sometimes understood in terms of technical debt.” (Valerdi, 2014)

COMMUNICATIONS

In its simplest form, communication is the transfer for data, information, or knowledge from one entity to another entity, however it is an extremely difficult activity to perform:

“Communications is never perfect and complete. Such a thing is not even possible. Even assuming for a moment that you, yourself, know what you intend, the receivers of the communication must jump across a gap at some point and must jump it all on their own. People with similar experience can jump a large gap, working even from mumblings and gestures. The more different another person is from you, the smaller the communication gap that he can jump. You have to back up, explain basic concepts, and then work forward until he builds his own bridge of experience and understands what you are saying. There is no end to this backing up. No matter how much you back up, there is always someone who will not understand.” (Cockburn, 2007)
Multiple organizational knowledge theories recognize two major types of knowledge: explicit and tacit.

“Explicit knowledge refers to the knowledge that can be translated into formal, systematic language. It is knowledge that can be written, documented, and widely distributed. Tacit knowledge is considered to be that which you know, but have difficulty explaining. It is often called ‘hidden knowledge’ because it is difficult to explicate, such as explaining to someone how to ride a bicycle. Tacit knowledge has a personality quality that makes it hard to formalize. Therefore, it is deeply rooted in action and commitment to a very specific context…. Explicit knowledge is easy to articulate and verbalize, systematic and objective, rational and logical, digital, sequential, comes from the past, and free of context. By contrast, tacit knowledge is difficult to articulate and verbalize, subjective, linked to experience and emotions, analogue, simultaneous, refers to the present and context dependent. Therefore tacit knowledge is deeply rooted in action, procedures, routines, commitments, ideals, values, and emotions. From this assertion, it follows that tacit knowledge includes technical-expert elements as well as cognitive ones. In other words, it involves the skills, experience and capabilities, mental models and precepts.” (Salmador & Florin, 2013)

Table 2 provides a brief summary of the SECI model which describes various modes for the transfer and conversion of tacit and explicit knowledge.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Definition</th>
<th>Conversion Modes</th>
</tr>
</thead>
<tbody>
<tr>
<td>S – Socialization</td>
<td>The process of converting new tacit knowledge through shared experiences</td>
<td>Tacit-to-Tacit Knowledge Conversion</td>
</tr>
<tr>
<td>E – Externalization</td>
<td>The process of articulating tacit knowledge into explicit knowledge</td>
<td>Tacit-to-Explicit Knowledge Conversion</td>
</tr>
<tr>
<td>C – Combination</td>
<td>The process of converting explicit knowledge into more complex and system sets of explicit knowledge</td>
<td>Explicit-to-Explicit Knowledge Conversion</td>
</tr>
<tr>
<td>I – Internalization</td>
<td>The process of embodying explicit knowledge into tacit knowledge</td>
<td>Explicit-to-Tacit Knowledge Conversion</td>
</tr>
</tbody>
</table>

It is argued that innovation is largely based upon the continuous exchange between tacit and explicit knowledge (Salmador & Florin, 2013).

In developing a framework for collaborative knowledge creation, Salisbury assesses the skill levels of various people within a team. As shown in Figure 5, novices rely more on explicit forms of knowledge, such as documentation and instructions, whereas experts use more tacit forms of knowledge, such as obtaining expert advice, to obtain the knowledge they require (Salisbury, 2008).
The manifesto has one value statement and two principles related to communications, of which the strongest principle states “The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.” (Manifesto for agile software development, 2001) As shown in Figure 6, face-to-face communications has multiple advantages. First, the information and knowledge exchange is enriched with non-verbal clues which gives additional insight into the level and confidence of the information exchange. It also allows each participant to ask and respond to questions in a very interactive dialog. The
conversation topic is generally restricted to the information required for a specific contest, thus the information exchange rate has low latency. However, there are also disadvantages. First, the communication pathways are synchronous, meaning both parties must be present at the same time for the exchange or information. Critical knowledge could quickly be lost if a critical team member leaves the project. Next, the number of possible communication pathways rapidly increases as more members are added to team. Thus, there is no guarantee that the same information will be consistently communicated to all team members. Face-to-face communications allows one team member to draw on the knowledge, experience, and perceptions of another team member, but access to that knowledge may be impeded or filter by many factors, such as a person’s ability recall from memory, his personality to willingly share information, and/or his cultural background and language abilities to conduct a meaningful information exchange.

![Figure 7: Face-To-Face Communications in a Group Setting](image)

As shown in Figure 7, face-to-face communications in a group setting brings additional dynamics to the information exchange. A group setting allows for the greatest access to the entire diverse set of information within the team. If the group setting is conducted in a professional manner, it can build trust promote shared learning throughout the team. However, group meetings can become very disruptive based on the personality of a single team member. Likewise, if multiple strong personalities are present and a mediating entity is not identified, then group direction moving forward may not be reach. Group settings allow new information
to presented, or identify the need to research new information pathways. Consequently, it is quite possible that multiple group meetings may be required to successfully arrive at a conclusion. Building and documenting team consensus is not an easy process and tools are being developed to support this process. For example, MITRE has developed an Agile Capability Mashup Environment (ACME) that uses low cost tools, such as horizontal whiteboards, cut outs, and webcams, to quickly develop, communicate, and document in an explicit knowledge format, project level knowledge outcomes based on individual knowledge inputs (Hall & Weiss, 2016).

Needless to say, both tacit and explicit forms of communications are required for a project to be successful.

**DOCUMENTATION**

In reviewing the literature, the phrase “comprehensive documentation” is directly associated with the concept of big up-front design (BUFD) where all elements of a system’s design are completely thought out and documented before any software coding begins. This massive amount of documentation then quickly falls out of sync soon after development starts (Erdogmus, 2009). As previously discussed, agile teams need to be concerned about the system’s architecture but “what is architecture” versus “what is design” may not be obvious.

“What does a particular project or organization mean by architecture? The concept has fuzzy boundaries. In particular, not all design is architecture. Agreeing on a definition is a useful exercise and a good starting point….Do not dilute the meaning of the term architecture by applying it to everything in sight. Not all design decisions are architectural. Few are, actually, and in many projects, they’re already made on day one.” (Abrahamsson et al., 2010)

Grady Booch offers these insights:

“As I’ve often said, the code is the truth, but not the whole truth, meaning that there are certain architectural decisions that cannot easily be discerned in the code itself. This is so because such decisions are manifested as mechanisms that are either scattered or tangled throughout the code, their meaning and presence are in the heads of the code’s creators and not easily evident by staring at it (the code)….It’s these bits of architectural decisions that are best documented elsewhere, external to the code base. Such decisions often live in tribal memory, in the heads of people. This is fine when the team is small, but when the system grows to economic significance, tribal memory is a particularly noisy and inefficient repository of architecture.

The architectural mechanisms that are not baked into the code and thus are in the heads are the things you want to (a) take time to document and, where possible, (b) create a domain-specific language that is baked into the code to implement it. My experience is that every reasonably software-intensive system will have a couple dozen such architectural mechanisms…..These are the kinds of decisions that can be documented in a static document of two or three dozen pages—any longer and no one will read it….this artifact becomes a vehicle for orienting new
folks to the code based as well as attending to some degree of architectural governance, whose simple goal is getting people to continue to grow the system according to those architecture principles.” (Booch, 2011)

Alistair Cockburn shares similar advice:

“the designer’s job is not pass along ‘the design’ but to pass along ‘the theories’ driving the design. The latter goal is more useful and more appropriate. It also highlights that knowledge of the theory is tacit in the owning, and so passing along the theory requires passing along both explicit and tacit knowledge.”

Cockburn then promotes a Theory Building View of a system and then summarizes by providing these recommendations for what should be put into documentation:

“That which helps the next programmer build an adequate theory of a program. This is enormously important. The purpose of the documentation is to jog memories in the reader, set up relevant pathways of thought about experiences and metaphors. This sort of documentation is more stable over the life of the program than just naming the pieces of system currently in place. The designers are allowed to use whatever forms of expression are necessary to set up those relevant pathways….Experienced designers often start their documentation with just:

- The metaphors
- Text describing the purpose of each major component
- Drawings of the major interactions between the major components

These three items alone take the next team a long way to constructing a useful theory of the design….Documentation cannot—and so need not—say everything. Its purpose is to help the next programmer build an accurate theory about the system.” (Cockburn, 2007)

For a client who had incurred tremendous amounts of technical debt caused by the absence of credible explicit knowledge and about his technical systems, the author was asked to identify, in priority order, those items that need to be explicitly described and the following was recommended:

1. Any knowledge that defines to users, operators, and maintainers how to operate and maintain the system
2. Any knowledge that describes how to rebuild and redeploy all the system, should a disaster occur
3. Any knowledge that is used to verify that system that been successfully rebuilt and redeployed such that it can again support the business
4. Any knowledge that allows future personnel to modify the system over its life cycle

PEOPLE

The manifesto places tremendous reliance on the skills, qualities, and talents of the individuals that comprise an agile team. Team members must be motivated, trustworthy, interactive,
reflective, able to work at a constant pace indefinitely, and collaboratively work within a self-organizing team. Boehm and Turner describe the:

“critical factors for agile methods include amicability, talent, skill and communication… and that both (agile and plan-driven methods) operate best with a mix of developer skills and understanding, but agile methods tend to need a richer mix of higher-skilled people… The plan-driven methods of course do better with great people, but are generally able to plan the project and architect the software so that less-capable people can contribute with low risk.” (Boehm & Turner, 2004)

Vidgen and Wang define that “self-organization is the ability of interconnected autonomous agents of a complex adaptive system to evolve into an organized form without external force.” (Vidgen & Wang, 2009) Glenda Eoyang provides a variance on this definition by stating that it is a “process by which the internal dynamics of a system generates system-wide patterns” but also adds that the system must be pushed away from thermodynamic equilibrium into a significant nonequilibrium region to require change. She identifies the following three necessary conditions of self-organization:

- Container – a bounding condition that distinguishes a system from its environment
- Significant difference – a distinction within a system that establishes a potential generative tension, which represents the potential for change
- Transforming exchange – A transfer of information, resources, or energy among system agents that results in changes within the agents and/or changes in system-wide patterns (Eoyang, 2001).

It should be highlighted that self-organization may not always occur for goodness, but simply represents internal dynamics that generates system-wide patterns.

For agile teams, the container is the value-added capabilities which the customer is requesting. Significant difference is closely related to the motivations of people and Cockburn discusses the following three potential intrinsic motivators for agile team members:

- Pride-in-work
- Pride-in-accomplishment
- Pride-in-contribution (Cockburn, 2007).

In the realm of transforming exchange, Kristina Grumadaite identified multiple factors that affect knowledge sharing in a self-organized system. She groups these factors based on culture, personal, and organizational characteristics (Grumadaite, 2013). Takpuie and Tanner developed a theoretical model shown in Figure 8 that defines and links various characteristics needed by Scrum team members to successfully transfer tacit knowledge during an agile software project (Takpuie & Tanner, 2016).
CONCLUSIONS

The following conclusions can be obtained from this paper:

- Projects can experience a progressive set of warning signs indicating project stress
- Agile methods can address these warning signs by placing requirements prioritization at the beginning of project, and then only committing the team to deliver a release that is progressively constructed through daily tasking and demonstrated at the end of periodic iterations.
- Because agile methods assign time-boxes to the delivery of features, an inherent risk in agile methods is the introduction and accumulation of technical debt
- To ensure technical quality is not reduced the following recommendations are provided:
  - Provide a definition of done for features/stories, iterations/sprints, and releases.
  - Expand the definition customer to ensure that non-functional requirements and long term project life cycle concerns are addressed.
  - Incorporate architecture concerns into the project such that non-functional and future functional requirements can be anticipated.
Communication mechanisms must incorporate both tacit and explicit knowledge mechanisms, realizing that innovative teams effectively implement both mechanisms.

When developing documentation, focus on those artifacts that provide others with the architectural constructs of the system. The software code does not represent the theory of the system but future developers need this awareness as they continue to modify the system over its life cycle.

Agile methods place significant reliance on the skills, qualities, and talent of the individuals that comprise the team. A model for the characteristics of a successful agile team member is provided and should be used in the identification and selection of people for inclusion in agile projects.

REFERENCES


