DACS State-of-the-Art/Practice Report

Agile Software Development

By
David Cohen, Mikael Lindvall, and Patricia Costa
Fraunhofer Center Maryland

Acknowledgements:
We would like to recognize our expert contributors who participated in the first eWorkshop on Agile Methods and thereby contributed to the section on State-of-the-Practice: Scott Ambler (Ronin International, Inc.), Ken Auer (RoleModel Software, Inc), Kent Beck (founder and director of the Three Rivers Institute), Winsor Brown (University of Southern California), Alistair Cockburn (Humans and Technology), Hakan Erdogmus (National Research Council of Canada), Peter Hantos (Xerox), Philip Johnson (University of Hawaii), Bil Kleb (NASA Langley Research Center), Tim Mackinnon (Connextra Ltd.), Joel Martin (National Research Council of Canada), Frank Maurer (University of Calgary), Atif Memon (University of Maryland and Fraunhofer Center for Experimental Software Engineering), Granville (Randy) Miller, (TogetherSoft), Gary Pollice (Rational Software), Ken Schwaber (Advanced Development Methods, Inc. and one of the developers of Scrum), Don Wells (ExtremeProgramming.org), William Wood (NASA Langley Research Center). We also would like to thank our collogues who helped arrange the eWorkshop and co-authored that same section: Victor Basili, Barry Boehm, Kathleen Dangle, Forrest Shull, Roseanne Tesoriero, Laurie Williams, and Marvin Zelkowitz.
We would like to thank Jen Dix for proof reading this report.

1 Introduction

The pace of life is more frantic than ever. Computers get faster every day. Start-ups rise and fall in the blink of an eye. And we stay connected day and night with our cable modems, cell phones, and Palm Pilots. Just as the world is changing, so too is the art of software engineering as practitioners attempt to keep in step with the turbulent times, creating processes that not only respond to change but embrace it.

These so-called Agile Methods are creating a buzz in the software development community, drawing their fair share of advocates and opponents. The purpose of this report is to address this interest and provide a comprehensive overview of the current State-of-the-Art as well as State-of-the-Practice for Agile Methods. As there is already much written about the motivations and aspirations of Agile Methods (e.g. (Abrahamsson et al., 2002)), we will emphasize the latter. The first section discusses the history behind the trend as well as the Agile Manifesto, a statement from the leaders of the Agile movement (Beck et al., 2001). The second section represents the State-of-the-Art and examines what it means to be Agile, discusses the role of management, describes and compares some of the more popular methods, provides a guide for deciding where an Agile approach is applicable, and lists common criticisms of Agile techniques. The third section represents State-of-the-Practice and summarizes empirical studies, anecdotal reports, and lessons learned. The report concludes with an Appendix that includes a detailed analysis of various Agile Methods for the interested reader.

The target audiences for this report include practitioners, who will be interested in the discussion of the different methods and their applications, researchers who may want to focus on the empirical studies and lessons learned, and educators looking to teach and learn more about Agile Methods.

It is interesting to note that there is a lack of literature describing projects where Agile Methods failed to produce good results. There are a number of studies reporting poor projects due to a negligent implementation of an Agile method, but none where practitioners felt they executed properly but the method failed to deliver on its promise. This may be a result of a reluctance to publish papers on unsuccessful projects, or it may in fact be an indication that, when implemented correctly, Agile Methods work.

1.1 History

Agile Methods are a reaction to traditional ways of developing software and acknowledge the ”need for an alternative to documentation driven, heavyweight software development processes” (Beck et al., 2001). In the implementation of traditional methods, work begins with the elicitation and documentation of a “complete” set of requirements, followed by architectural and high-level design, development, and inspection. Beginning in the mid-1990s, some practitioners found these initial development steps frustrating and, perhaps, impossible (Highsmith, 2002a). The industry and technology move too fast, requirements “change at rates that swamp traditional methods” (Highsmith et al., 2000), and customers have become increasingly unable to definitively state their needs up front while, at the same time, expecting more from their software. As a result, several consultants have

independently developed methods and practices to respond to the inevitable change they were experiencing. These Agile Methods are actually a collection of different techniques (or practices) that share the same values and basic principles. Many are, for example, based on iterative enhancement, a technique that was introduced in 1975 (Vic Basili and Turner, 1975).

In fact, most of the Agile practices are nothing new (Cockburn and Highsmith, 2001a). It is instead the focus and values behind Agile Methods that differentiate them from more traditional methods. Software process improvement is an evolution in which newer processes build on the failures and successes of the ones before them, so to truly understand the Agile movement, we need to examine the methods that came before it.

According to Beck, the Waterfall Method (Royce, 1970) came first, as a way in which to assess and build for the users' needs. It began with a complete analysis of user requirements. Through months of intense interaction with users and customers, engineers would establish a definitive and exhaustive set of features, functional requirements, and non-functional requirements. This information is well-documented for the next stage, design, where engineers collaborate with others, such as database and data structure experts, to create the optimal architecture for the system. Next, programmers implement the well-documented design, and finally, the complete, perfectly designed system is tested and shipped (Beck, 1999a).

This process sounds good in theory, but in practice it did not always work as well as advertised. Firstly, users changed their minds. After months, or even years, of collecting requirements and building mockups and diagrams, users still were not sure of what they wanted – all they knew was that what they saw in production was not quite "it." Secondly, requirements tend to change mid-development and when requirements are changed, it is difficult to stop the momentum of the project to accomodate the change. The traditional methods may well start to pose difficulties when change rates are still relatively low (Boehm, 2002) because programmers, architects, and managers need to meet, and copious amounts of documentation need to be kept up to date to accommodate even small changes (Boehm, 1988). The Waterfall model was supposed to fix the problem of changing requirements once and for all by freezing requirements and not allowing any change, but practitioners found that requirements just could not be pinned down in one fell swoop as they had anticipated (Beck, 1999a).

Incremental and iterative techniques focusing on breaking the development cycle into pieces evolved from the Waterfall model (Beck, 1999a), taking the process behind waterfall and repeating it throughout the development lifecycle. Incremental development aimed to reduce development time by breaking the project into overlapping increments. As with the Waterfall model, all requirements are analyzed before development begins; however, the requirements are then broken into increments of stand-alone functionality. Development of each increment may be overlapped, thus saving time through concurrent "multitasking" across the project.

While incremental development looked to offer timesavings, evolutionary methods like iterative development and the Spiral Model (Boehm, 1988) aimed to better handle changing requirements and manage risk. These models assess critical factors in a structured and planned way at multiple points in the process rather than trying to mitigate them as they appear in the project.

Iterative development breaks the project into iterations of variable length, each producing a complete deliverable and building on the code and documentation produced before it. The first iteration starts with the most basic deliverable, and each subsequent iteration adds the next logical set of features. Each piece is its own waterfall process beginning with analysis, followed by design, implementation, and finally testing. Iterative development deals well with change, as the only complete requirements necessary are for the current iteration. Although tentative requirements need to exist for the next iteration, they do not need to be set in stone until the next analysis phase. This approach allows for changing technology or the customer to change their mind with minimal impact on the project's momentum.

Similarly, the Spiral Model avoids detailing and defining the entire system upfront. Unlike iterative development, however, where the system is built piece by piece prioritized by functionality, Spiral prioritizes requirements by risk. Spiral and iterative development offered a great leap in agility over the Waterfall process, but some practitioners believed that they still did not respond to change as nimbly as necessary in the evolving business world. Lengthy planning and analysis phases as well as a sustained emphasis on extensive documentation kept projects using iterative techniques from being truly Agile, in comparison with today's methods.

Another important model to take into account in these discussions is the Capability Maturity Model (CMM[1]) (Paulk, 1993), "a five-level model that describes good engineering and management practices and prescribes improvement priorities for software organizations" (Paulk, 2001). The model defines 18 key process areas and 52 goals for an organization to become a level 5 organization. Most software organizations' maturity level is 'Chaotic' (CMM level one) and only a few are 'Optimized' (CMM level five). CMM focuses mainly on large projects and large organizations, but can be tailored to fit small as well as large projects due to the fact that it is formulated in a very general way that fits diverse organizations' needs. The goals of CMM are to achieve process consistency, predictability, and reliability (Paulk, 2001).

Ken Schwaber was one practitioner looking to better understand the CMM-based traditional development methods. He approached the scientists at the DuPont Chemical's Advanced Research Facility posing the question: "Why do the defined processes advocated by CMM not measurably deliver?" (Schwaber, 2002). After analyzing the development processes they returned to Schwaber with some surprising conclusions. Although CMM focuses on turning software development into repeatable, defined, and

[1] We use the terms CMM and SW-CMM interchangeable to denote the Software CMM from the Software Engineering Institute (SEI).

predictable processes, the scientists found that many of them were, in fact, largely unpredictable and unrepeatable because (Schwaber, 2002):

- "Applicable first principles are not present
- The process is only beginning to be understood
- The process is complex
- The process is changing and unpredictable"

Schwaber, who would go on to develop Scrum, realized that to be truly Agile, a process needs to accept change rather than stress predictability (Schwaber, 2002). Practitioners came to realize that methods that would respond to change as quickly as it arose were necessary (Turk et al., 2002), and that in a dynamic environment, "creativity, not voluminous written rules, is the only way to manage complex software development problems" (Cockburn and Highsmith, 2001a).

Practitioners like Mary Poppendieck and Bob Charette[2] also began to look to other engineering disciplines for process inspiration, turning to one of the more innovate industry trends at the time, Lean Manufacturing. Started after World War II by Toyoda Sakichi, its counter-intuitive practices did not gain popularity in the United States until the early 1980s. While manufacturing plants in the United States ran production machines at 100% and kept giant inventories of both products and supplies, Toyoda kept only enough supplies on hand to run the plant for one day, and only produced enough products to fill current orders. Toyoda also tightly integrated Dr. W. Edwards Deming's Total Quality Management philosophy with his process. Deming believed that people inherently want to do a good job, and that managers needed to allow workers on the floor to make decisions and solve problems, build trust with suppliers, and support a "culture of continuous improvement of both process and products" (Poppendieck, 2001). Deming taught that quality was a management issue and while Japanese manufacturers were creating better and cheaper products, United States manufacturers were blaming quality issues on their workforce (Poppendieck, 2001).

Poppendieck lists the 10 basic practices which make Lean Manufacturing so successful, and their application to software development:

1. Eliminate waste – eliminate or optimize consumables such as diagrams and models that do not add value to the final deliverable.
2. Minimize inventory – minimize intermediate artifacts such as requirements and design documents.
3. Maximize flow – use iterative development to reduce development time.
4. Pull from demand – support flexible requirements.
5. Empower workers – generalize intermediate documents, "tell developers what needs to be done, not how to do it."
6. Meet customer requirements – work closely with the customer, allowing them to change their minds.
7. Do it right the first time – test early and refactor when necessary.

[2] Bob Charette's "Lean Development" method will be discussed later.

8. Abolish local optimization – flexibly manage scope.
9. Partner with suppliers – avoid adversarial relationships, work towards developing the best software.
10. Create a culture of continuous improvement – allow the process to improve, learn from mistakes and successes (Poppendieck, 2001).

Independently, Kent Beck rediscovered many of these values in the late 1990s when he was hired by Chrysler to save their failing payroll project, Chrysler Comprehensive Compensation (C3). The project was started in the early 1990s as an attempt to unify three existing payroll systems (The C3 Team, 1998) and had been declared a failure when Beck arrived. Beck, working with Ron Jeffries (Highsmith et al., 2000), decided to scrap all the existing code and start the project over from scratch. A little over a year later, a version of C3 was in use and paying employees. Beck and Jeffries were able to take a project that had been failing for years and turn it around 180 degrees. The C3 project became the first project to use eXtreme Programming (Highsmith et al., 2000) (discussed in detail later) relying on the same values for success as Poppendiek's Lean Programming.

Similar stories echo throughout the development world. In the early 1990s, the IBM Consulting Group hired Alistair Cockburn to develop an object-oriented development method (Highsmith et al., 2000). Cockburn decided to interview IBM development teams and build a process out of best practices and lessons learned. He found that "team after successful team 'apologized' for not following a formal process, for not using high-tech [tools], for 'merely' sitting close to each other and discussing while they went," while teams that had failed followed formal processes and were confused why it hadn't worked, stating "maybe they hadn't followed it well enough" (Highsmith et al., 2000). Cockburn used what he learned at IBM to develop the Crystal Methods (discussed in detail later).

The development world was changing and, while traditional methods were hardly falling out of fashion, it was obvious that they did not always work as intended in all situations. Practitioners recognized that new practices were necessary to better cope with changing requirements. And these new practices must be people-oriented and flexible, offering "generative rules" over "inclusive rules" which break down quickly in a dynamic environment (Cockburn and Highsmith, 2001a). Highsmith and Cockburn summarize the new challenges facing the traditional methods:

- Satisfying the customer has taken precedence over conforming to original plans
- Change will happen – the focus is not how to prevent it but how to better cope with it and reduce the cost of change throughout the development process
- "Eliminating change early means being unresponsive to business conditions – in other words, business failure"
- "The market demands and expects innovative, high quality software that meets its needs – and soon" (Cockburn and Highsmith, 2001a).

1.2 The Agile Manifesto

"[A] bigger gathering of organizational anarchists would be hard to find" Beck stated, (Beck et al., 2001) when seventeen of the Agile proponents came together in early 2001

to discuss the new software developments methods. "What emerged was the Agile 'Software Development' Manifesto. Representatives from Extreme Programming, SCRUM, DSDM, Adaptive Software Development, Crystal, Feature-Driven Development, Pragmatic Programming, and others sympathetic to the need for an alternative to documentation driven, heavyweight software development processes convened" (Beck et al., 2001). They summarized their viewpoint, saying that "the Agile movement is not anti-methodology, in fact, many of us want to restore credibility to the word methodology. We want to restore a balance. We embrace modeling, but not in order to file some diagram in a dusty corporate repository. We embrace documentation, but not hundreds of pages of never-maintained and rarely used tomes. We plan, but recognize the limits of planning in a turbulent environment" (Beck et al., 2001). The manifesto itself reads as follows.

> "We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value
>
> - Individuals and interaction over process and tools
> - Working software over comprehensive documentation
> - Customer collaboration over contract negotiation
> - Responding to change over following a plan
>
> That is, while there is a value in the items on the right, we value the items on the left more" (Beck et al., 2001).

The Manifesto has become an important piece of the Agile Movement in that it characterizes the values of Agile methods and how Agile distinguishes itself from traditional methods. Glass amalgamates the best of the Agile and traditional approaches by analyzing the Agile manifesto and comparing it with traditional values (Glass, 2001).

On *Individuals and interaction over process and tools*: Glass believes that the Agile community is right on this point: "Traditional software engineering has gotten too caught up in its emphasis on process" (Glass, 2001). At the same time "most practitioners already know that people matter more than process" (Glass, 2001).

On *Working software over comprehensive documentation*: Glass agrees with the Agile community on this point too, although with some caveat: "It is important to remember that the ultimate result of building software is product. Documentation matters…but over the years, the traditionalists made a fetish of documentation. It became the prime goal of the document-driven lifecycle" (Glass, 2001).

On *Customer collaboration over contract negotiation*: Glass sympathies with both sides regarding this statement: "I deeply believe in customer collaboration, and … without it nothing is going to go well. I also believe in contracts, and I would not undertake any significant collaborative effort without it" (Glass, 2001)

On *Responding to change over following a plan*: Both sides are right regarding this statement, according to Glass: Over they years, we have learned two contradictory lessons: 1. "[C]ustomers and users do not always know what they want at the outset of a software project, and we must be open to change during project execution" (Glass, 2001) 2. "Requirement change was one of the most common causes of software project failure" (Glass, 2001).

This view, that both camps can learn from each other, is commonly held, as we will see in the next section.

1.3 Agile and CMM(I)

As mentioned above, Agile is a reaction against traditional methodologies, also known as rigorous or plan-driven methodologies (Boehm, 2002). One of the models often used to represent traditional methodologies is the Capability Maturity Model (CMM[3]) (Paulk, 1993) and its replacement[4] CMMI, an extension of CMM based on the same values[5]. Not much has been written about CMMI yet, but we believe that for this discussion, what is valid for CMM is also valid for CMMI[6].

As mentioned above, the goals of CMM are to achieve process consistency, predictability, and reliability. Its proponents claim that it can be tailored to also fit the needs of small projects even though it was designed for large projects and large organizations (Paulk, 2001).

Most Agile proponents do not, however, believe CMM fits their needs at all: "If one were to ask a typical software engineer whether the Capability Maturity Model for Software and process improvement were applicable to Agile Methods, the response would most likely range from a blank stare to a hysterical laughter" (Turner and Jain, 2002). One reason is that "CMM is a belief in software development as a defined process …[that] can be defined in detail, [that] algorithms can be defined, [that] results can be accurately measured, and [that] measured variations can be used to refine the processes until they are repeatable within very close tolerances" (Highsmith, 2002b). "For projects with any degree of exploration at all, Agile developers just do not believe these assumptions are valid. This is a deep fundamental divide – and not one that can be reconciled to some comforting middle ground" (Highsmith, 2002b).

Many Agile proponents also dislike CMM because of its focus on documentation instead of code. A "typical" example is the company that spent two years working (not using CMM though) on a project until they finally declared it a failure. Two years of working resulted in "3,500 pages of use cases, an object model with hundreds of classes, thousands of attributes (but no methods), and, of course, no code" (Highsmith, 2002b).

---

[3] We use the terms CMM and SW-CMM interchangeable to denote the Software CMM from the Software Engineering Institute (SEI).

[4] CMM will be replaced by CMMI; see "How Will Sunsetting of the Software CMM® Be Conducted" at http://www.sei.cmu.edu/cmmi/adoption/sunset.html

[5] The Personal Software Process (PSP) (Humphrey, 1995) is closely related to the CMM.

[6] E-mail conversation with Sandra Shrum, SEI.

The same document-centric approach resulting in "documentary bloat that is now endemic in our field" (DeMarco and Boehm, 2002) is also reported by many others.

While Agile proponents see a deep divide between Agile and traditional methods, this is not the case for proponents of traditional methods. Mark Paulk, the man behind CMM, is, for example, surprisingly positive about Agile Methods and claims that "Agile Methods address many CMM level 2 and 3 practices" (Paulk, 2002). XP[7], for example, addresses most level 2[8] and 3[9] practices, but not level 4 and 5 (Paulk, 2001).

As a matter of fact, "most XP projects that truly follow the XP rules and practices could easily be assessed at CMM level 2 if they could demonstrate having processes for the following:" (Glazer, 2001)

- "Ensuring that the XP Rules and Practices are taught to new developers on the project
- Ensuring that the XP Rules and Practices are followed by everyone
- Escalating to decision makers when the XP Rules and Practices are not followed and not resolved within the project
- Measuring the effectiveness of the XP Rules and Practices
- Providing visibility to management via appropriate metrics from prior project QA experience
- Knowing when the XP Rules and Practices need to be adjusted
- Having an independent person doing the above" (Glazer, 2001).

Glazer adds, "with a little work on the organizational level, CMM level 3 is not far off" (Glazer, 2001).

So according to some, XP and CMM *can* live together (Glazer, 2001), at least in theory. One reason is that we can view XP as a software development methodology and CMM as a software management methodology. CMM tells us *what* to do, while XP tells us *how* to do it.

Others agree that there is no conflict. Siemens, for example, does not see CMM and Agility as a contradiction. Agility has become a necessity with increasing market pressure, "but should be built on top of an appropriately mature process foundation, not instead of it" (Paulisch and Völker, 2002). Many make a distinction between "turbulent" environments and "placid" environments, and conclude that CMM is not applicable to the "turbulent" environments. These claims are based on misconceptions: "In fact, working

[7] As XP is the most documented method, it often is used as a representative sample of Agile Methods.
[8] XP supports the following Level 2 practices according to (Paulk, 2001): *requirements management*, *software project planning*, *software project tracking and oversight, software quality assurance*, and *software configuration management*, but not *software subcontract management*
[9] XP supports the following Level 3 practices according to (Paulk, 2001): *organization process focus*, *organization process definition*, *software product engineering*, *inter-group coordination*, and *peer reviews*. XP does not support the level 3 practices *training program* and *integrated software management*.

under time pressure in the age of agility requires even better organization of the work than before!" (Paulisch and Völker, 2002).

Regarding the criticism about heavy documentation in CMM projects, Paulk replies: "over-documentation is a pernicious problem in the software industry, especially in the Department of Defense (DoD) projects" (Paulk, 2002). "[P]lan-driven methodologists must acknowledge that keeping documentation to a minimum useful set is necessary. At the same time, "practices that rely on tacit knowledge[10]…may break down in larger teams…" (Paulk, 2002). Others claim that "CMM does not require piles of process or project documentation" and there are "various organizations that successfully can manage and maintain their process with a very limited amount of paper" (Paulisch and Völker, 2002).

CMMI is the latest effort to build maturity models and consists of Process Areas (PA) and Generic Practices (GP). CMMI is similar to CMM, but more extensive in that it covers the discipline of system engineering. In an attempt to compare Agile and CMMI, Turner analyzed their values and concluded that their incompatibilities are overstated and that their strengths and weaknesses complement each other (Turner and Jain, 2002).

While many tired of traditional development techniques are quick to show support for the Agile movement, often as a reaction against CMM, others are more skeptical. A common criticism, voiced by Steven Rakitin, views Agile as a step backwards from traditional engineering practices, a disorderly "attempt to legitimize the hacker process" (Rakitin, 2001). Where processes such as Waterfall and Spiral stress lengthy upfront planning phases and extensive documentation, Agile Methods tend to shift these priorities elsewhere. XP, for example, holds brief iteration planning meetings in the Planning Game to prioritize and select requirements, but generally leaves the system design to evolve over iterations through refactoring, resulting in hacking (Rakitin, 2001). This accusation of Agile of being no more than hacking is frenetically fought (Bowers, 2002) and in response to this criticism, Beck states: "Refactoring, design patterns, comprehensive unit testing, pair programming – these are not the tools of hackers. These are the tools of developers who are exploring new ways to meet the difficult goals of rapid product delivery, low defect levels, and flexibility" (Highsmith et al., 2000). Beck says, 'the only possible values are 'excellent' and 'insanely excellent' depending on whether lives are at stake or not'… You might accuse XP practitioners of being delusional, but not of being poor-quality-oriented hackers" (Highsmith et al., 2000). "Those who would brand proponents of XP or SCRUM or any of the other Agile Methodologies as 'hackers' are ignorant of both the methodologies and the original definition of the term hacker" (Beck et al., 2001). In response to the speculation that applying XP would result in a Chaotic development process (CMM level 1), one of the Agile proponents even concluded that "XP is in some ways a 'vertical' slice through the levels 2 through 5" (Jeffries, 2000).

The question whether Agile is hacking is probably less important than whether Agile and CMM(I) can co-exist. This is due to the fact that many organizations need both to be

[10] Agile Methods rely on undocumented (tacit) knowledge and avoid documentation.

Agile and show that they are mature enough to take on certain contracts. A model that fills that need and truly combines the Agile practices and the CMM key processes has not, that we are aware of, been developed yet.

2 State-of-the-Art

This section discusses what it means to be Agile, describes a selected set of Agile Methods, and concludes with a discussion on whether an organization is ready to adopt Agile Methods.

2.1 What does it mean to be Agile?

The goal of Agile Methods is to allow an organization to be agile, but what does it mean to be Agile? Jim Highsmith says that being Agile means being able to "Deliver quickly. Change quickly. Change often" (Highsmith et al., 2000). While Agile techniques vary in practices and emphasis, they share common characteristics, including iterative development and a focus on interaction, communication, and the reduction of resource-intensive intermediate artifacts. Developing in iterations allows the development team to adapt quickly to changing requirements. Working in close location and focusing on communication means teams can make decisions and act on them immediately rather than wait on correspondence. Reducing intermediate artifacts that do not add value to the final deliverable means more resources can be devoted to the development of the product itself and it can be completed sooner; "A great deal of the Agile movement is about what I would call 'programmer power'" (Glass, 2001). These characteristics add maneuverability to the process (Cockburn, 2002), whereby an Agile project can identify and respond to changes more quickly than a project using a traditional approach.

Cockburn and Highsmith discuss the Agile "world view," explaining "what is new about Agile Methods is not the practices they use, but their recognition of people as the primary drivers of project success, coupled with an intense focus on effectiveness and maneuverability" (Cockburn and Highsmith, 2001a). Practitioners agree that being Agile involves more than simply following guidelines that are supposed to make a project Agile. True agility is more than a collection of practices; it's a frame of mind. Andrea Branca states, "other processes may *look* Agile, but they won't *feel* Agile" (Cockburn, 2002).

2.2 A selection of Agile Methods

Agile Methods have much in common, such as what they value, but they also differ in the practices they suggest. In order to characterize different methods, we will examine the following Agile Methods: Extreme Programming, Scrum, Crystal Methods, Feature Driven Development, Lean Development, and Dynamic Systems Development Methodology. We will attempt to keep the depth and breadth of our discussion consistent for each method, though it will naturally be limited by the amount of material available; XP is well-documented and has a wealth of available case studies and reports, while DSDM is subscription-based, making it much more difficult to find information. The other methods lie somewhere in between. See also Appendix A, in which we attempt to further analyze these methods.

2.2.1 Extreme Programming

Extreme Programming is undoubtedly the hottest Agile Method to emerge in recent years. Introduced by Beck and Jeffries, et al. in 1998 (The C3 Team, 1998) and further popularized by Beck's Extreme Programming Explained: Embrace Change in 1999 and

numerous articles since, XP owes much of its popularity to developers disenchanted with traditional methods (Highsmith, 2002a) looking for something new, something extreme.

The 12 rules of Extreme Programming, true to the nature of the method itself, are concise and to the point. In fact, you could almost implement XP without reading a page of Beck's book.

- *The Planning Game*: At the start of each iteration customers, managers, and developers meet to flesh out, estimate, and prioritize requirements for the next release. The requirements are called "user stories" and are captured on "story cards" in a language understandable by all parties.
- *Small Releases*: An initial version of the system is put into production after the first few iterations. Subsequently, working versions are put into production anywhere from every few days to every few weeks.
- *Metaphor*: Customers, managers, and developers construct a metaphor, or set of metaphors after which to model the system.
- *Simple Design*: Developers are urged to keep design as simple as possible, "say everything once and only once" (Beck, 1999a).
- *Tests*: Developers work test-first; that is, they write acceptance tests for their code before they write the code itself. Customers write functional tests for each iteration and at the end of each iteration, all tests should run.
- *Refactoring*: As developers work, the design should be evolved to keep it as simple as possible.
- *Pair Programming*: Two developers sitting at the same machine write all code.
- *Continuous Integration:* Developers integrate new code into the system as often as possible. All functional tests must still pass after integration or the new code is discarded.
- *Collective ownership*: The code is owned by all developers, and they may make changes anywhere in the code at anytime they feel necessary.
- *On-site customer*: A customer works with the development team at all times to answer questions, perform acceptance tests, and ensure that development is progressing as expected.
- *40-hour Weeks*: Requirements should be selected for each iteration such that developers do not need to put in overtime.
- *Open Workspace*: Developers work in a common workspace set up with individual workstation around the periphery and common development machines in the center.

Practitioners tend to agree that the strength of Extreme Programming does not result from each of the 12 practices alone, but from the emergent properties arising from their combination. Highsmith lists five key principles of XP, all of which are enhanced by its practices: communication, simplicity, feedback, courage, and quality work (Highsmith, 2002a).

Practitioners of XP clearly state where the model works and where it does not.

Team size: Because the development team needs to be co-located, team size is limited to the number of people that can fit in a single room, generally agreed to be from 2-10.

Iteration length: XP has the shortest recommended iteration length of the Agile Methods under consideration, 2 weeks.

Support for distributed teams: Because of XP's focus on community and co-location, distributed teams are not supported.

System criticality: XP is not necessarily geared for one system or another. However, most agree that there is nothing in XP itself that should limit its applicability.

2.2.2 Scrum

Scrum, along with XP, is one of the more widely used Agile Methods. Ken Schwaber first described Scrum in 1996 (Schwaber, 2002) as a process that "accepts that the development process is unpredictable," formalizing the "do what it takes" mentality, and has found success with numerous independent software vendors. The term is borrowed from Rugby: "[A] Scrum occurs when players from each team huddle closely together…in an attempt to advance down the playing field" (Highsmith, 2002a).

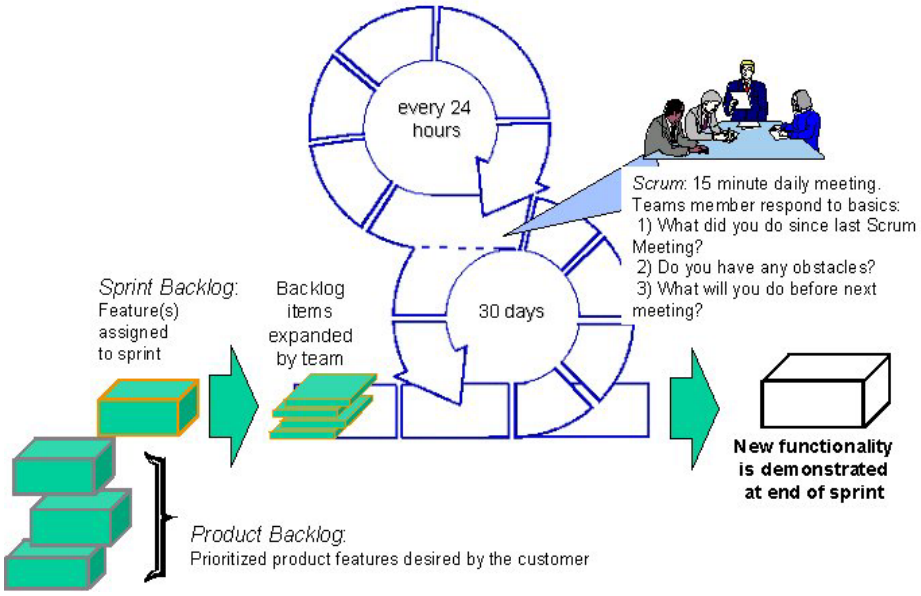

Figure 1: The Scrum Lifecycle, from controlchaos.com

Figure 1 depicts the Scrum lifecycle. Scrum projects are split into iterations, "sprints" consisting of the following:

Pre-sprint planning: All work to be done on the system is kept in what is called the "release backlog." During the pre-sprint planning, features and functionality are selected from the release backlog and placed into the "sprint backlog," or a prioritized collection

of tasks to be completed during the next sprint. Since the tasks in the backlog are generally at a higher level of abstraction, pre-sprint planning also identifies a Sprint Goal reminding developers why the tasks are being performed and at which level of detail to implement them (Highsmith, 2002a).

Sprint: Upon completion of the pre-sprint planning, teams are handed their sprint backlog and "told to sprint to achieve their objectives" (Schwaber, 2002). At this point, tasks in the sprint backlog are frozen and remain unchangeable for the duration of the sprint. Team members choose the tasks they want to work on and begin development. Short daily meetings are critical to the success of Scrum. Scrum meetings are held every morning to enhance communication and inform customers, developers, and managers on the status of the project, identify any problems encountered, and keep the entire team focused on a common goal.

Post-sprint meeting: After every sprint, a post-sprint meeting is held to analyze project progress and demonstrate the current system.

Schwaber summarizes the key principles of Scrum:

- "Small working teams that maximize communication, minimize overhead, and maximize sharing of tacit, informal knowledge
- Adaptability to technical or marketplace (user/customer) changes to ensure the best possible product is produced
- Frequent 'builds', or construction of executables, that can be inspected, adjusted, tested, documented, and built on
- Partitioning of work and team assignments into clean, low coupling partitions, or packets
- Constant testing and documentation of a product as it is built
- Ability to declare a product 'done' whenever required (because the competition just shipped, because the company needs the cash, because the user/customer needs the functions, because that was when it was promised…)" (Schwaber, 2002).

Team size: Development personnel are split into teams of up to seven people. A complete team should at least include a developer, quality assurance engineer, and a documenter.

Iteration length: While Schwaber originally suggested sprint lengths from 1 to 6 weeks (Schwaber, 2002), durations are commonly held at 4 weeks (Highsmith, 2002a).

Support for distributed teams: While Scrum's prescription does not explicitly mention distributed teams or provide built-in support; a project may consist of multiple teams that could easily be distributed.

System criticality: Scrum does not explicitly address the issue of criticality.

2.2.3 The Crystal Methods

The Crystal Methods were developed by Alistair Cockburn in the early 1990s. He believed that one of the major obstacles facing product development was poor communication and modeled the Crystal Methods to address these needs. Cockburn explains his philosophy: "To the extent that you can replace written documentation with face-to-face interactions, you can reduce the reliance on written work products and improve the likelihood of delivering the system. The more frequently you can deliver running, tested slices of the system, the more you can reduce the reliance on written 'promissory' notes and improve the likelihood of delivering the system" (Highsmith et al., 2000). Highsmith adds: "[Crystal] focuses on people, interaction, community, skills, talents, and communication as first order effects on performance. Process remains important, but secondary" (Highsmith, 2002a).

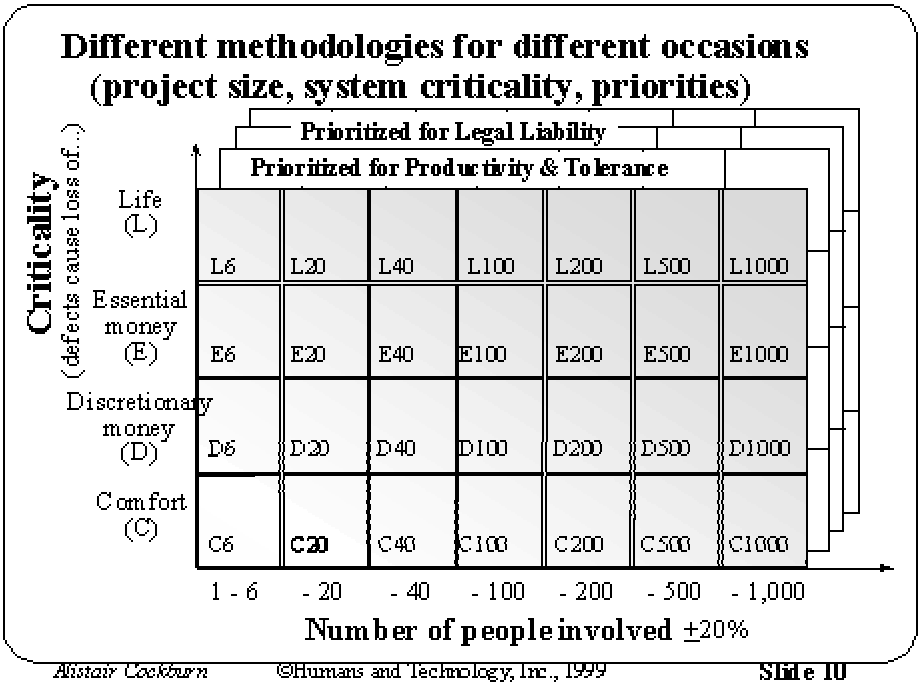

Figure 2: Crystal Methods Framework, from crystalmethodologies.org

Cockburn's methods are named "crystal" to represent a gemstone, each facet is another version of the process, all arranged around an identical core (Highsmith, 2002a). As such, the different methods are assigned colors arranged in ascending opacity; the most Agile version is Crystal Clear, followed by Crystal Yellow, Crystal Orange, Crystal Red, etc. The version of crystal you use depends on the number of people involved, which translates into a different degree of emphasis on communication.

As you add people to the project, you translate right on the graph in Figure 2 to more opaque versions of crystal. As project criticality increases, the methods "harden" and you move upwards on the graph. The methods can also be altered to fit other priorities, such as productivity or legal liability.

All Crystal methods begin with a core set of roles, work products, techniques, and notations, and this initial set is expanded as the team grows or the method hardens. As a necessary effect, more restraints leads to a less Agile method but Highsmith stresses that they are Agile nonetheless because of a common mindset (Highsmith, 2002a).

Team size: the Crystal Family accommodates any team size; however, Cockburn puts a premium on premium people.

Iteration length: Up to 4 months for large, highly critical projects.

Support for distributed teams: Crystal Methodologies have built in support for distributed teams.

System criticality: Crystal supports 4 basic criticalities: failure resulting in loss of comfort, discretionary money, essential money, and life.

2.2.3.1 Feature Driven Development

Feature Driven Development arose in the late 1990s from a collaboration between Jeff DeLuca and Peter Coad. Their flagship project, like XP's C3 Project, was the Singapore Project. DeLuca was contracted to save a failing, highly complicated lending system. The previous contractor had spent two years producing over 3,500 pages of documentation but no code (Highsmith, 2002a). DeLuca started from the beginning and hired Coad to assist with the object modeling. Combining their previous experiences, they developed the feature-oriented development approach that came to be known as FDD.

Highsmith explains FDD's core values:

- "A system for building systems is necessary in order to scale to larger projects
- A simple, well-defined process works best
- Process steps should be logical *and their worth immediately obvious to each team member*
- 'Process pride' can keep the real work from happening
- Good processes move to the background so the team members can focus on results
- Short, iterative, feature-driven life cycles are best" (Highsmith, 2002a).

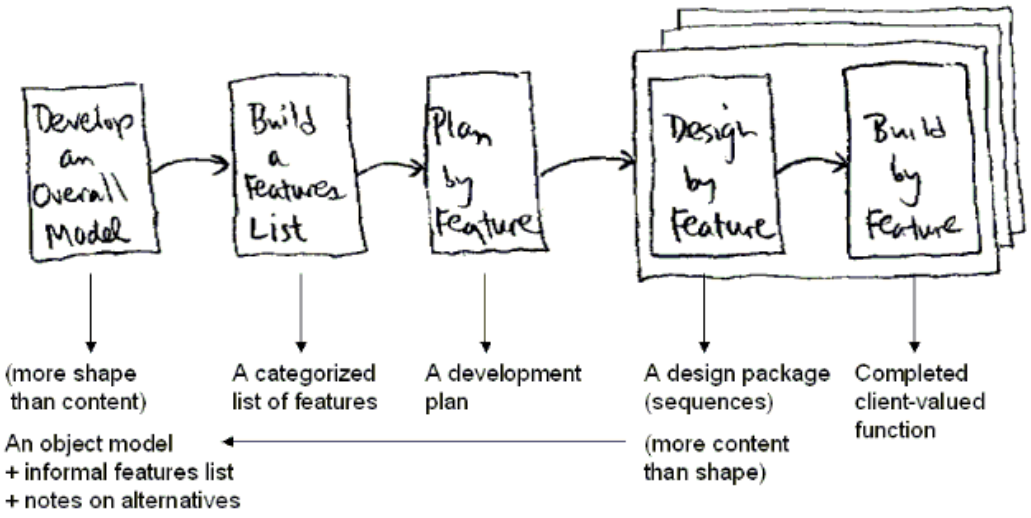

Figure 3: FDD Process, from togethercommunity.com

Develop an overall model: As depicted in figure 3, the FDD process begins with developing a model. Team members and experts work together to create a "walk-though" version of the system.

Build a features list: Next, the team identifies a collection of features representing the system. Features are small items useful in the eyes of the client. They are similar to XP story cards written in a language understandable by all parties. Features should take up to 10 days to develop (Highsmith, 2002a). Features requiring more time than 10 days are broken down into sub features.

Plan by feature: The collected feature list is then prioritized into subsections called "design packages." The design packages are assigned to a chief programmer, who in turn assigns class ownership and responsibility to the other developers.

Design by feature & build by feature: After design packages are assigned, the iterative portion of the process begins. The chief programmer chooses a subset of features that will take 1 to 2 weeks to implement. These features are then planned in more detail, built, tested, and integrated.

Team size: Team size varies depending on the complexity of the feature at hand. DeLuca stresses the importance of premium people, especially modeling experts.

Iteration length: Up to two weeks.

Support for distributed teams: FDD is designed for multiple teams and while it does not have built-in support for distributed environments, it should be adaptable.

Criticality: The FDD prescription does not specifically address project criticality.

2.2.4 Lean Development

Lean Development (LD), started by Bob Charette, draws on the success Lean Manufacturing found in the automotive industry in the 1980s. While other Agile Methods look to change the development process, Charette believes that to be truly Agile you need to change how companies work from the top down. Lean Development's 12 principles focus on management strategies:

1. Satisfying the customer is the highest priority
2. Always provide the best value for the money
3. Success depends on active customer participation
4. Every LD project is a team effort
5. Everything is changeable
6. Domain, not point, solutions
7. Complete, do not construct
8. An 80 percent solution today instead of 100 percent solution tomorrow
9. Minimalism is essential
10. Needs determine technology
11. Product growth is feature growth, not size growth
12. Never push LD beyond its limits (Highsmith, 2002a)

Because LD is more of a management philosophy than a development process, team size, iteration length, team distribution, and system criticality are not directly addressed.

2.2.4.1 Dynamic Systems Development Method

Dynamic Systems Development Method, according to their website[11], is not so much a method as it is a framework. Arising in the early 1990s, DSDM is actually a formalization of RAD practices (Highsmith, 2002a). As depicted in Figure 4, the DSDM lifecycle has six stages: Pre-project, Feasibility Study, Business Study, Functional Model Iteration, Design and Build Iteration, Implementation, and Post-project.

[11] http://www.dsdm.org

Figure 4: The DSDM Lifecycle, from www.dsdm.org

Pre-project: The pre-project phase establishes that the project is ready to begin, funding is available, and that everything is in place to commence a successful project.

Feasibility study: DSDM stresses that the feasibility study should be short, no more than a few weeks (Stapleton, 1997). Along with the usual feasibility activities, this phase should determine whether DSDM is the right approach for the project.

Business study: The business study phase is "strongly collaborative, using a series of facilitated workshops attended by knowledgeable and empowered staff who can quickly pool their knowledge and gain consensus as to the priorities of the development" [dsdm.org]. The result of this phase is the Business Area Definition, which identifies users, markets, and business processes affected by the system.

Functional Model Iteration: The functional model iteration aims to build on the high-level requirements identified in the business study. The DSDM framework works by building a number of prototypes based on risk and evolves these prototypes into the complete system. This phase and the design and build phase have a common process:

1. "Identify what is to be produced
2. Agree how and when to do it
3. Create the product
4. Check that it has been produced correctly (by reviewing documents, demonstrating a prototype or testing part of the system)"[12]

Design and build iteration: The prototypes from the functional model iteration are completed, combined, and tested and a working system is delivered to the users.

[12] http://www.dsdm.org

Implementation: During this phase, the system is transitioned into use. An Increment Review Document is created during implementation that discusses the state of the system. Either the system is found to meet all requirements and can be considered complete, or there is missing functionality (due to omission or time concerns). If there is still work to be done on the system, the functional model, design and build, and implementation phases are repeated until the system is complete.

Post-project: This phase includes normal post-project clean up as well as on-going maintenance.

Because of DSDM's framework nature, it does not specifically address team size, exact iteration lengths, distribution, or system criticality.

2.2.5 Agile Modeling

Agile Modeling (AM) is proposed by Scott Ambler (Ambler, 2002a). It is a method based on values, principles and practices that focus on modeling and documentation of software. AM recognizes that modeling is a critical activity for a project success and addresses how to model in an effective and Agile manner (Ambler, 2002b).

The three main goals of AM are:

1. "To define and show how to put into practice a collection of values, principles and practices that lead to effective and lightweight modeling
2. To address the issue on how to apply modeling techniques on Agile software development processes.
3. To address how you can apply effective modeling techniques independently of the software process in use" (Ambler, 2002b).

AM is not a complete software development method, instead it focuses only on documentation and modeling and can be used with any software development process. You start with a base process and tailor it to use AM. Ambler illustrates, for example, how to use AM with both XP and Unified Process (UP) (Ambler, 2002a).

The values of AM include those of XP – communication, simplicity, feedback and courage – and include also humility. It is critical for project success that your have effective communication in your team and also with the stakeholder of the project. You should strive to develop the simplest solution that meets your needs and to get feedback often and early. You should also have the courage to make and stick to your decisions and also have the humility to admit that you may not know everything and that others may add value to your project efforts.

Following is a summary of the principles of AM:

1. Assume simplicity: Assume the simplest solution is the best solution.
2. Content is more important than representation: You can use "post it" notes, whiteboard or a formal document, what matters is the content.
3. Embrace change: Accept the fact the change happens.

4. Enabling your next effort is your secondary goal: Your project can still be a failure if you deliver it and it is not robust enough to be extended.
5. Everyone can learn from everyone else: Recognize that you can never truly master something, there is always an opportunity to learn from others.
6. Incremental change: Change your system a small portion at a time, instead of trying to get everything accomplished in one big release.
7. Know your models: You need to know the strengths and weaknesses of models to use them effectively.
8. Local Adaptation: You can modify AM to adapt to your environment.
9. Maximize stakeholder investment: Stakeholders have the right to decide how to invest their money and they should have a final say on how those resources are invested.
10. Model with a purpose: If you cannot identify why you are doing something, why bother?
11. Multiple models: You have a variety of modeling artifacts (e.g. UML diagrams, data models, user interface models, etc.).
12. Open and honest communication: Open and honest communications enable people to make better decisions.
13. Quality work: You should invest effort into making permanent artifacts (e.g. code, documentation) of sufficient quality.
14. Rapid feedback: Prefer rapid feedback to delayed feedback whenever possible.
15. Software is your primary goal: The primary goal is to produce high-quality software that meets stakeholders' needs.
16. Travel light: Create just enough models and documents to get by.
17. Work with people's instincts: Your instincts can offer input into your modeling efforts (Ambler, 2002a).

Here is a summary of the AM practices:

1. Active Stakeholder participation: Project success requires a significant level of stakeholders involvement.
2. Apply modeling standards: Developers should agree and follow a common set of modeling standards on a software project.
3. Apply the right artifact(s): Modeling artifacts (UML diagram, use case, data flow diagram, source code) have different strengths and weaknesses. Make sure you use the appropriate one for your situation.
4. Collective ownership: Everyone can modify any model and artifact they need to.
5. Consider testability: When modeling, always ask the question: "how are we going to test this?"
6. Create several models in parallel: By creating several models you can iterate between them and select the best model that suits your needs.
7. Create simple content: You should not add additional aspects to your artifacts unless they are justifiable.
8. Depict models simply: Use a subset of the modeling notation available to you, creating a simple model that shows the key features you are trying to understand.
9. Discard temporary models: Discard working models created if they no long add value to your project.

10. Display models publicly: Make your models accessible for the entire team.
11. Formalize contract models: A contract model is always required when you are working with an external group that controls and information resource (e.g. a database) required by your system.
12. Iterate to another artifact: Whenever you get "stuck" working on an artifact (if you are working with a use case and you are struggling to describe the business logic), iterate with another artifact.
13. Model in small increments: Model a little, code a little, test a little and deliver a little.
14. Model to communicate: One of the reasons to model is to communicate with the team or to create a contract model.
15. Model to understand: The main reasons for modeling is to understand the system you are building, consider approaches and choose the best one.
16. Model with others: It is very dangerous to model alone.
17. Prove it with code: To determine if your model will actually work, validate your model by writing the corresponding code.
18. Reuse existing resources: There is a lot of information available that modelers can reuse to their benefit.
19. Update only when it hurts: Only update a model or artifact when you absolutely need to.
20. Use the simplest tools: Use the simplest tool that works in your case: a napkin, a whiteboard and even CASE tools if they are the most effective for your situation (Ambler, 2002a).

Since AM is not a complete software process development method and should be used with other development methods, the team size, exact iteration lengths, distribution and system criticality will depend on the development process being used.

2.3 Characteristics of selected Agile Methods

The table below presents the collected prescriptive characteristics of the discussed methods.

Table 1. Prescriptive Characteristics

	XP	Scrum	Crystal	FDD	LD	DSDM	AM
Team Size	2-10	1-7	Variable	Variable	N/A		
Iteration Length	2 weeks	4 weeks	< 4 months	< 2 weeks			
Distributed Support	No	Adaptable	Yes	Adaptable			
System Criticality	Adaptable	Adaptable	All types	Adaptable			

As we have seen, different Agile Methods have different characteristics. A brief comparison of Crystal Clear and XP resulted, for example, in the following:

- XP pursues greater productivity through increased discipline, but it is harder for a team to follow
- Crystal Clear permits greater individuality within the team and more relaxed work habits in exchange for some loss in productivity
- Crystal Clear may be easier for a team to adopt, but XP produces better results if the team can follow it
- A team can start with Crystal Clear and move to XP; a team that fails with XP can move to Crystal Clear (Highsmith et al., 2000)

2.4 Is your organization ready for Agile Methods?

As we have seen, there are many Agile Methods to select from, each bringing practices that will change the daily work of the organization. Before an organization selects and implements an Agile Method it should ponder whether or not it is ready for Agile or not. Scott Ambler discusses factors affecting successful adoption in his article *When Does(n't) Agile Modeling Make Sense?* (Ambler, 2002d). Number one on his list: "Agile adoption will be most successful when there is a conceptual fit between the organization and the Agile view. Also important for adoption are your project and business characteristics: Is your team already working incrementally? What is the team's motivation? What kind of support can the team expect?" (Ambler, 2002d). Are there adequate resources available? How volatile are the project requirements? Barry Boehm suggests using traditional methods for projects where requirements change less than 1% per month (Boehm, 2002).

Ambler also suggests the importance of an Agile champion – someone to tackle the team's challenges so they can work easily (Ambler, 2002d). Boehm stresses the importance of having well trained developers, since Agile processes tend to place a high degree of reliance on a developer's tacit knowledge (Boehm, 2002). The customer also needs to be devoted to the project, and must be able to make decisions: "Poor customers result in poor systems" (Cockburn and Highsmith, 2001a). Boehm adds, "Unless customer participants are committed, knowledgeable, collaborative, representative, and empowered, the developed products typically do not transition into use successfully, even though they may satisfy the customer" (Boehm, 2002).

Alistair Cockburn lists a few caveats when adopting an Agile process:

- As the number of people on a project grows, there is an increased strain on communications
- As system criticality increases, there is decreased "tolerance for personal stylistic variations"

If Agile Methods do not seem to be a good fit for your project or organization right off the bat, Ambler suggests partial adoption (Ambler, 2002d). Look at your current development process, identify the areas that need the most improvement, and adopt Agile techniques that specifically address your target areas. After successful adoption of the chosen practices and, even better, a demonstrated improvement to your overall process, continue selecting and implementing Agile techniques until you have adopted the entire process.

3 State-of-the-Practice

Agile Methods are gaining popularity in industry although they comprise a mix of accepted and controversial software engineering practices. In recent years, there have been many stories and anecdotes of industrial teams experiencing success with Agile Methods. There is, however, an urgent need to empirically assess the applicability of these methods, in a structured manner, in order to build an experience base for better decision-making. In order to reach their goals, software development teams need, for example, to understand and choose the right models and techniques to support their projects. They must consider key questions such as: What is the best life-cycle model to choose for a particular project? What is an appropriate balance of effort between documenting the work and getting the product implemented? When does it pay-off to spend major efforts on planning in advance and avoid change, and when is it more beneficial to plan less rigorously and embrace change?

While previous sections of this report discussed Agile Methods from a state-of-the-art perspective, this section addresses these questions and captures the state-of-the-practice and the experiences from applying Agile Methods in different settings. The section starts with results from an eWorkshop on Agile Methods followed by other empirical studies.

3.1 eWorkshop on Agile Methods

The goal of the Center for Empirically-Based Software Engineering (CeBASE) is to collect and disseminate knowledge on software engineering. A central activity toward achieving this goal has been the running of "eWorkshops" (or on-line meetings). The CeBASE project defined the eWorkshop (Basili et al., 2001) and has, for example, used it to collect empirical evidence on defect reduction and COTS (Shull et al., 2002). This section is based on a paper that discusses the findings of an eWorkshop in which experiences and knowledge were gathered from and shared between Agile experts located throughout the world (Lindvall et al., 2002a). The names of these 18 participants are listed in the acknowledgements of this report.

3.1.1 Seeding the eDiscussion

For this eWorkshop, Barry Boehm's January 2002 IEEE Computer article (Boehm, 2002), Highsmith and Cockburn's articles (Highsmith and Cockburn, 2001) (Cockburn and Highsmith, 2001b), and the Agile Manifesto[13] served as background material together with material defining Agile Methods, such as Extreme Programming (XP) (Beck, 1999b), Scrum (Schwaber, 2002), Feature Driven Development (FDD) (Coad et al., 1999), Dynamic Systems Development Method (DSDM) (Stapleton, 1997), Crystal (Cockburn, 2000), and Agile modeling (Ambler, 2002a).

Boehm brings up a number of different characteristics regarding Agile Methods compared to what he calls "Plan-Driven Methods," the more traditional Waterfall, incremental or Spiral methods. Boehm contends that Agile, as described by Highsmith and Cockburn (Highsmith and Cockburn, 2001), emphasizes several critical people-factors, such as amicability, talent, skill, and communication, at the same time noting that

[13] http://www.agileAlliance.org

49.99% of the world's software developers are below average in these areas. While Agile does not require uniformly highly capable people, it relies on tacit knowledge to a higher degree than plan-driven projects that emphasize documentation. Boehm argues that there is a risk that this situation leads to architectural mistakes that cannot be easily detected by external reviewers due to the lack of documentation (Boehm, 2002).

Boehm also notes that Cockburn and Highsmith conclude that "Agile development is more difficult for larger teams" and that plan-driven organizations scale-up better (Highsmith and Cockburn, 2001). At the same time, the bureaucracy created by plan-driven processes does not fit small projects either. This again, ties back to the question of selecting the right practices for the task at hand (Boehm, 2002).

Boehm questions the applicability of the Agile emphasis on simplicity. XP's philosophy of YAGNI (You Aren't Going to Need It) (Beck, 1999b) is a symbol of the recommended simplicity that emphasizes eliminating architectural features that do not support the current version. Boehm feels this approach fits situations where future requirements are unknown. In cases where future requirements are known, the risk is, however, that the lack of architectural support could cause severe architectural problems later. This raises questions like: What is the right balance between creating a grandiose architecture up-front and adding features as they are needed? Boehm contends that plan-driven processes are most needed in high-assurance software (Boehm, 2002). Traditional goals of plan-driven processes such as predictability, repeatability, and optimization, are often characteristics of reliable safety critical software development. Knowing for what kind of applications different practices (traditional or Agile) are most beneficial is crucial, especially for safety critical applications where human lives can be at stake if the software fails.

Based on background material, the following issues were discussed:

1. The definition of Agile
2. Selecting projects suitable for Agile
3. Introducing the method
4. Managing the project.

Each of these will be discussed in the following section. The full discussion summary can be found on the FC-MD web site (http://fc-md.umd.edu).

3.1.2 Definition

The eWorkshop began with a discussion regarding the definition of Agile and its characteristics, resulting in the following working definition.
Agile Methods are:

- Iterative: Delivers a full system at the very beginning and then changes the functionality of each subsystem with each new release
- Incremental: The system as specified in the requirements is partitioned into small subsystems by functionality. New functionality is added with each new release
- Self-organizing: The team has the autonomy to organize itself to best complete the work items.

- Emergent: Technology and requirements are "allowed" to emerge through the product development cycle.

All Agile Methods follow the four values and 12 principles of the Agile Manifesto.

3.1.3 Selecting Projects Suitable for Agile Methods

The most important factor that determines when Agile is applicable is probably project size. From the discussion it became clear that there is:

- Plenty of experience of teams of up to 12 people
- Some descriptions of teams of approximately 25 people
- A few data points regarding teams of up to 100 people, e.g. 45 & 90-person teams
- Isolated descriptions of teams larger than 100 people. (e.g. teams of 150 and 800 people were mentioned and documented in (Highsmith, 2002a).

Many participants felt that any team could be Agile, regardless of its size. Alistair Cockburn argued that size is an issue. As size grows, coordinating interfaces becomes a dominant issue. Face-to-face communication breaks down and becomes more difficult and complex past 20-40 people. Most participants agreed, but think that this statement is true for any development process. Past 20-40 people, some kind of scale-up strategies must be applied.

One scale-up strategy that was mentioned was the organization of large projects into teams of teams. In one occasion, an 800-person team was, for example, organized using "scrums of scrums" (Schwaber and Beedle, 2002). Each team was staffed with members from multiple product lines in order to create a widespread understanding of the project as a whole. Regular, but short, meetings of cross-project sub-teams (senior people or common technical areas) were held regularly to coordinate the project and its many teams of teams. It was pointed out that a core team responsible for architecture and standards (also referred to as glue) is needed in order for this configuration to work. These people work actively with the sub-teams and coordinate the work.

Effective ways of coordinating multiple teams include yearly holding conferences to align interfaces, rotation of people between teams in 3-month internships, and shared test case results. Examples of strategies for coping with larger teams are documented in Jim Highsmith's Agile Software Development Ecosystems (Highsmith, 2002a), in which the 800-person team is described.

There is an ongoing debate about whether or not Agile requires "good people" to be effective. This is an important argument to counter since "good people" can make just about anything happen and that specific practices are not important when you work with good people. This suggests that perhaps the success of Agile Methods could be attributed to the teams of good folks, rather than practices and principles. On the other hand, participants argued that Agile Methods are intrinsically valuable. Participants agreed that a certain percentage of experienced people are needed for a successful Agile project. There was some consensus that 25%-33% of the project personnel must be "competent and experienced."

"Competent" in this context means:

- Possess real-world experience in the technology domain
- Have built similar systems in the past
- Possess good people and communication skills

It was noted that experience with actually building systems is much more important than experience with Agile development methods. The level of experience might even be as low as 10% if the teams practice pair programming (Williams et al., 2000) and if the makeup of the specific programmers in each pair is fairly dynamic over the project cycle (termed "pair rotation"). Programmers on teams that practice pair rotation have an enhanced environment for mentoring and for learning from each other.

One of the most widespread criticisms of Agile Methods is that they do not work for systems that have criticality, reliability and safety requirements. There was some disagreement about suitability for these types of projects. Some participants felt that Agile Methods work if performance requirements are made explicit early, and if proper levels of testing can be planned for. Others argue that Agile best fits applications that can be built "bare bones" very quickly, especially applications that spend most of their lifetime in maintenance.

There was also some disagreement about the best Agile Methods for critical projects. A consensus seemed to form that the Agile emphasis on testing, particularly the test-driven development practice of XP, is the key to working with these projects. Since all tests have to be passed before release, projects developed with XP can adhere to strict (or safety) requirements. Customers can write acceptance tests that measure nonfunctional requirements, but they are more difficult and may require more sophisticated environments than JUnit tests.

Many participants felt that Agile Methods render it easier to address critical issues since the customer gives requirements, makes important issues explicit early and provides continual input. The phrase "responsibly responding to change" implies that there is a need to investigate the source of the change and adjust the solution accordingly, not just respond and move on. When applied right, "test first" satisfies this requirement.

3.1.4 Introducing Agile Methods: Training requirements

An important issue is how to introduce Agile Methods in an organization and how much formal training is required before a team can start using it. A majority (though not all) of the participants felt that Agile Methods require less formal training than traditional methods. For example, pair programming helps minimize what is needed in terms of training, because people mentor each other. This kind of mentoring (by some referred to as tacit knowledge transfer) is argued to be more important than explicit training. The emphasis is rather on skill development, not on learning Agile Methods. Training on how to apply Agile Methods can many times occur as self-training. Some participants have seen teams train themselves successfully. The participants concluded that there should be enough training material available for XP, Crystal, Scrum, and FDD.

3.1.5 Project management: Success factors and Warning signs

One of the most effective ways to learn from previous experience is to analyze past projects from the perspective of success factors. The three most important success factors identified among the participants were culture, people, and communication.

To be Agile is a cultural matter. If the culture is not right, then the organization cannot be Agile. In addition, teams need some amount of local control; they must have the ability to adapt working practices as they feel appropriate. The culture must also be supportive of negotiation as negotiation forms a large part of Agile culture.

As discussed above, it is important to have competent team members. Organizations using Agile use fewer, but more competent people. These people must be trusted, and the organization must be willing to live with the decisions developers make, not consistently second-guess their decisions.

Organizations that want to be agile need to have an environment that facilitates rapid communication between team members. Examples are physically co-located teams and pair programming.

It was pointed out that organizations need to carefully implement these success factors in order for them to happen. The participants concluded that Agile Methods are most appropriate when requirements are emergent and rapidly changing (and there is always some technical uncertainty!). Fast feedback from the customer is another factor that is critical for success. In fact, Agile is based on close interaction with the customer and expects that the customer will be on-site to provide the quickest possible feedback, a critical success factor.

A critical part of project management is recognizing early warning signs that indicate that something has gone wrong. The question posed to participants was: How can management know when to take corrective action to minimize risks?

Participants concluded that the daily meetings provide a useful way of measuring problems. As a result of the general openness of the project and because discussions of these issues are encouraged during the daily meeting, people will bring up problems. Low morale expressed by the people in the daily meeting will also reveal that something has gone wrong that the project manager must deal with. Another indicator is when "useless documentation" is produced, even though it can be hard to determine what useless documentation is. Probably the most important warning sign is when the team is falling behind on planned iterations. As a result, having frequent iterations is very important to monitor for this warning sign.

A key tenet of Agile Methods (especially in XP) is refactoring. Refactoring means improving the design of existing code without changing the functionality of the system. The different forms of refactoring involve: simplifying complex statements, abstracting common solutions into reusable code, and the removal of duplicate code.

Not all participants were comfortable with refactoring the architecture of a system because refactoring would affect all internal and external stakeholders. Instead, frequent refactoring of reasonably sized code, and minimizing its scope to keep changes more local, were recommended. Most participants felt that large-scale refactoring is not a problem, since it is frequently necessary and more feasible using Agile Methods. Participants strongly felt that traditional "Big Design Up Front (BDUF)" is rarely on target, and its lack of applicability is often not fed back to the team that created the BDUF, making it impossible for them to learn from experience. It was again emphasized that testing is the major issue in Agile. Big architectural changes do not need to be risky, for example, if a set of automated tests is provided as a "safety net."

Product and project documentation is a topic that has drawn much attention in discussions about Agile. Is any documentation necessary at all? If so, how do you determine how much is needed? Scott Ambler commented that documentation becomes out of date and should be updated only "when it hurts." Documentation is a poor form of communication, but is sometimes necessary in order to retain critical information. Many organizations demand more documentation than is needed. Organizations' goal should be to communicate effectively, and documentation should be one of the last options to fulfill that goal. Barry Boehm mentioned that a project documentation makes it easier for an outside expert to diagnose problems. Kent Beck disagreed, saying that, as an outside expert who spends a large percentage of his time diagnosing projects, he is looking for people "stuff" (like quiet asides) and not technical details. Bil Kleb said that with Agile Methods, documentation is assigned a cost and its extent is determined by the customer (excepting internal documentation). Scott Ambler suggested his Agile Documentation essay as good reference for this topic (Ambler, 2001a).

3.2 Lessons Learned

Several lessons can be learned from this discussion that should prove be useful to those considering applying Agile Methods in their organization. These lessons should be carefully examined and challenged by future projects to identify the circumstances in which they hold and when they are not applicable.

Any team could be Agile, regardless of the team size, but should be considered because greater numbers of people make communication more difficult. Much has been written about small teams, but less information is available regarding larger teams, for which sale-up strategies are necessary.

- Experience is important for an Agile project to succeed, but experience with actually building systems is much more important than experience with Agile Methods. It was estimated that 25%-33% of the project personnel must be "competent and experienced", but the necessary percentage might even be as low as 10% if the teams practice pair programming due to the fact that they mentor each other.

- Agile Methods require less formal training than traditional methods. Pair programming helps minimize what is needed in terms of training, because people

mentor each other. Mentoring is more important than regular training that can many times be completed as self-training. Training material is available in particular for XP, Crystal, Scrum, and FDD.

- Reliable and safety-critical projects can be conducted using Agile Methods. Performance requirements must be made explicit early, and proper levels of testing must be planned. It is easier to address critical issues using Agile Methods since the customer gives requirements, makes sets explicit priorities early and provides continual input.

- The three most important success factors are culture, people, and communication. Agile Methods need cultural support otherwise they will not succeed. Competent team members are crucial. Agile Methods use fewer, but more competent people. Physically co-located teams and pair programming support rapid communication. Close interaction with the customer and frequent customer feedback are critical success factors.

- Early warning signs can be spotted in Agile projects, e.g. low morale expressed during the daily meeting. Other signs are production of "useless documentation" and delays of planned iterations.

- Refactoring should be done frequently and of reasonably sized code, keeping the scope down and local. Large-scale refactoring is not a problem, and is more feasible using Agile Methods. Traditional "BDUF" is a waste of time and doesn't lead to a learning experience. Big architectural changes do not need to be risky if a set of automated tests is maintained.

- Documentation should be assigned a cost and its extent be determined by the customer. Many organizations demand more than is needed. The goal should be to communicate effectively and documentation should be the last option.

In another eWorkshop, the following experiences were reported regarding Agile and CMM:

- At Boeing, XP was used before CMM was implemented and they were able to implement the spirit of the CMM without making large changes to their software processes. They used XP successfully, and CMM helped introduce the Project Management Discipline.
- ABB is introducing XP while transitioning from CMM to CMMI worldwide. They are in the opposite position from Boeing: CMM(I) was introduced several years before XP, which is true for their corporate research centers as well as for business units.
- NASA Langley Research Center reported a better match with CMM and Agile when the CMM part is worded generally, as in "follow a practice of choice", and not delving into specifics such as, "must have spec sheet 5 pages long."
- ABB added that their organization has adopted the CMMI framework and they are incorporating Agile practices into the evolutionary development lifecycle

model. They believe that there is a clear distinction between life cycle models and continuous process improvement models such as CMMI and both are not incompatible. No incompatibilities between Agile and CMM were reported (Lindvall et al., 2002b).

3.3 Case Studies

Another important source of empirical data is case studies. In this section we report from a selected number of case studies on different aspects of applying Agile Methods.

3.3.1 Introducing XP

Karlström reports on a project at Online Telemarketing in Lund, Sweden, where XP was applied (Karlström, 2002). The report is based both on observation and interviews with the team that applied XP. The project was a success despite the fact that the customer had a very poor idea of the system at the beginning of the project. All XP practices were practically introduced. The ones that worked the best were: planning game, collective ownership, and customer on site. They found small releases and testing difficult to introduce.

Online Telemarketing is a small company specializing in telephone-based sales of third party goods. It had recently been expanded internationally and management realized that a new sales support system would be required. COTS alternatives were investigated and discarded because they were expensive, and incorporating desired functionality was difficult. The lack of detailed requirements specification from management and the lack of a similar system motivated the use of XP.

The system was developed in Visual Basic, and it has 10K lines of code. The development started in December 2000 and the first functional system was launched in April 2001. The product has been in operation since August 2001.

The senior management at Online Telemarketing assumed the role of a customer. Configuration management started without a tool and developers were supposed to copy the files to a directory. This worked when they had two developers. When the team grew they add a text file to manage copies to checkout directory. This solution still presented problems when the developers were out or working in different schedules. Once the communication issues were resolved the solution worked.

The following experience were reported:

1. The planning game. In total 150 stories were implemented. Stories were added during the whole project. In the beginning, time estimates were inaccurate, but became better after a few weeks passed. Breaking the stories into tasks was hard for the developers, causing them to create too detailed stories. It was hard to set a common level of detail for the stories. In the end, this practice proved to be one of the greatest successes.
2. Small releases: The first iteration took too long because of the lack of experience with XP. Once a complete bare system was implemented, it was easier to implement small releases. During the long initial release, they tried to maintain

the communication between the developers and the customer, to avoid mistakes in development.

3. Metaphor: They used a document that was an attempt at a requirements document, before they decided to use XP and their metaphor. As the project progressed the document was not updated.
4. Simple design: The development team stressed implementing the simplest possible solution at all times. They thought that this practice saved them time when a much larger solution would be implemented, avoiding unnecessary code.
5. Testing: Test-first was difficult to implement at first and VBUnit was hard to learn and set up. When the time pressure increased, the developers started to ignore test-first. Although they saw the benefits, it involved too much work. Since it was hard to write tests for the GUI and the team thought that mastering a GUI testing tool would take too long, they decided to test the GUI manually. The customer tested the functionality of the system before each release, and when a problem was found a correction card was created.
6. Refactoring: No tools for refactoring were used, and the team performed minor refactoring continuously. No major refactoring of the code was performed.
7. Pair Programming: They used pair programming at all times. At first the developers were not comfortable, but later they started to work naturally and efficiently in pairs. The developers were inexperienced, which might be why they felt comfortable. The lead developer thought that they produced code faster in pairs than they would have if working alone.
8. Collective ownership: This practice worked well. The developers avoided irritations by thinking of bugs as group issues instead of as someone's defect. The configuration management, however, was not very effective and sometimes developers were afraid to change code if not in direct contact with others.
9. Continuous integration: this practice was natural in the development environment. As soon as the code was finished, it was integrated.
10. 40-hour week: Since the developers were part-time, this practice was adjusted and followed.
11. On site-customer: This practice worked well, despite some schedule conflicts because the developers were part-time and the customer was played by busy senior managers.
12. Coding standards: A coding standard document was developed in the beginning of the project and updated when needed. Over time, developers became a little relaxed in following the standards, but once this was identified as an issue, it was reinforced.

3.3.2 Launching XP at a Process-Intensive Company

Grenning reports experiences from the introduction of an adaptation of XP in an organization with a large formal software development process (Grenning, 2001). The task was to build a new system to replace an existing safety-critical legacy system. The new system was an embedded-systems application running on Windows NT.

The system was divided into subsystems developed by different units; the author was called to help one of these units. The author was very enthusiastic about XP and decided to convince the team to apply some of the techniques.

The company already had a process in place that added a lot of overhead to the development because requirements were partially defined and deadlines were tight.

Recognizing that the organization culture believed in up-front requirements and designs followed by reviews and approvals, the team decided to "choose their battles" and introduce the practices that would be most beneficial for the project. One major issue was documentation. How much documentation was sufficient? The team would be developing a piece that was supposed to work with pieces being developed by other teams using the standard process at the organization. They identified that they needed enough documentation to define the product requirements, sustain technical reviews and support the system's maintainers. Clean and understandable source code and some form of interface documentation, due to the need to collaborate with other teams. XP recognizes that documentation has a cost and that incomplete documentation might be cost-effective. But choosing not to create any documentation would be unacceptable in this environment.

When proposing the new approach to management, story cards appeared unacceptable and the team decided to use cases instead of story cards. The management was concerned with future maintenance of the system; if the system was transitioned to another team, more than readable code would be needed. After some discussions, the team decided to create high-level documentation at the end of the project, instead of documenting in the beginning followed by updates during the project. The management, however, still wanted to be able to review the design. The proposed solution was to document the design decisions and have them reviewed at the end of every month. This removed the review from the critical path of the project.

Despite compromising on a few issues, the team got permission to apply test-first, pair programming, short iterations, continuous integration, refactoring, planning, and team membership for the customer.

According to Grenning, at the project's conclusion, the programmers were happy with their creation. After the fourth iteration the project manager was satisfied. The reason is that they already had working code at a point when their regular process only would have produced only three documents. The project manager also recognized that dependencies between the features were almost non-existent since they followed the customer's priorities and not the priorities dictated by a big design up front. The team was Agile and able to adapt to other subsystems' changing needs.

Grenning points out the importance of including senior team members because they "spread the wealth of knowledge, and both they (senior people) and their pair partners learn" (Grenning, 2001). Despite the fact that the project was terminated due to changes

in the market, the management was very pleased with results and two other pilot projects were started.

At the end of the report, the author gives advice to management and developers willing to try XP. For managers, it is important to try XP on a team with open-minded leaders, encourage XP practices, and recruit a team that wants to try XP instead of forcing a team to use XP. For developers, the advice is to identify the problems that they might solve, develop a sales pitch and do a pilot project (Grenning, 2001).

3.3.3 Using XP in a Maintenance Environment

Poole and Huisman report their experience with introducing XP in Iona Technologies (Poole and Huisman, 2001). Because of its rapid growth and time-to-market pressures, the engineering team often ignored engineering practices. As a result, they ended up with a degenerated code that was salvaged in reengineering efforts that led to XP.

As part of the improvement effort, they used a bug-tracking tool to identify problem areas of the code. The code was cleaned through the elimination of used code and the introduction of patterns that made it easier to test, maintain and understand the code. As part of this effort, one lead engineer promoted stronger engineering practices making engineers constantly consider how they could improve the quality of their code. Testing of the whole system was also automated. After all these transformations the company saw a lot of improvement. Despite their progress, however, they still had issues to resolve regarding testing, visibility, morale and personal work practices. They already had a maintenance process in place that had a lot in common with XP practices, so they decided to apply XP in order to solve the remaining issues.

All bugs reported by customers, enhancement requirements, and new functional requirements are documented. That documentation is accessible by both customers and the team. They do not use index cards for the requirements and the requirements are not in the form of user stories yet. Index cards are used to track tasks and those tasks are added to storyboards. The developers estimate the tasks, and the customers prioritize them. When the tasks are finished, they are removed from the storyboard, recorded in a spreadsheet and the cards are archived in the task log. They also introduced daily stand-up meetings to increase visibility and also stimulate communication among the team members.

They automated their whole testing process, making it possible to test the whole system with the click of a button. All engineers are supposed to test the whole system after changes are made to ensure nothing was broken.

They report that convincing programmers to do pair programming is extremely difficult. Luckily, their pair programming experience came to them by accident. In 2000, a customer engineer working with them paired with the developers. The experience was good, the team felt that they worked more effectively, the overall productivity was high and morale improved. They are now trying to formally introduce pair programming.

Increments are kept short and they continuously produce small releases. Refactoring has also been extensively applied, which can be seen in the code reviews. Engineers are encouraged to identify areas of the code that are candidates for refactoring, and they follow up after delivery with a refactoring task in the storyboard. In order to improve communications, they also changed the workspace to make pair programming easier and facilitate discussions of their ideas on whiteboards.

The effort seemed to pay off and the productivity increase is noticeable. In their point of view the greatest benefit to the team has been the increase in visibility. The storyboards let people see what others are doing and help management track progress and plan.

They conclude the paper pointing out that the application of pair programming and collection of metrics can improve their process. They believe that improving the pair programming initiative can improve their lack of cross-training among the code base's many modules. The metrics are critical to the planning game, since estimating how long a story will take requires finding a similar story in the past and researching how long it took. Currently they are tracking estimates and actuals on a spreadsheet and are working to integrate this into their defect tracking system.

3.3.4 XP's "Bad Smells"

In an attempt to provide early warning signals ("bad smells") when applying XP, Elssamadisy and Schalliol analyze a three-year project involving 30 developers (50 in total) that produced about 500,000 lines of executable code (Elssamadisy and Schalliol, 2002). The project switched to XP due to previous experiences with ineffective traditional methods. The lessons learned from the experience of applying XP can be useful to others:

- Customers are typically happy during early iterations but later begin to complain about many things from all iterations. The customer needs to be coached to provide early and honest feedback. Elssamadisy and Schalliol suggest they think like buying a tailored suit in which you cannot just have measurements taken at the beginning
- Programmers are typically not sure of how functionality works together. Large complex systems require a good metaphor or overview
- Everyone claims the story cards are finished, yet it requires weeks of full-time development to deliver a quality application. The solution is to create a precise list of tasks that must be completed before a story is finished, and make sure programmers adhere to the rules: Acknowledge poorly estimated stories and reprioritize. Do not rush to complete them and cut corners with refactoring or testing.

The authors concluded pointing out that the team needs to be conscious of the process the whole time, and that laziness will affect the whole team.

3.3.5 Introducing Scrum in Organizations

The authors of this paper (Cohn and Ford, 2002) have successfully introduced Scrum to seven organizations over a period of four years. They discuss their lessons learned as well as mistakes.

In several cases they encountered resistance from developers who preferred to develop non-code artifacts and from those who "valued their contribution to a project by the number of meetings attended in a given day" (Cohn and Ford, 2002). Some even tried to put more documentation back into the process. The solution used by the authors is to not intervene and instead let peers decide whether to adopt suggestions or not.

The authors were surprised to find that many developers view Agile Methods as micromanagement. In traditional projects, developers meet the project manager once a week, but in an Agile environment they meet daily. To change developers' perception, the project manager has to show that he is there to remove obstacles, not to complain about missed deadlines and must not be judgmental when developers report that they will be delayed with their tasks.

Distributed development has been successful, and the authors believe that methods other than Scrum can also be used in distributed environments. They propose waiting two or three months until developers get used to Agile development before implementing distributed development. In order for distributed development to work, many people must be brought together for the first few weeks of the project.

Experience shows that Agile Methods require good developers and that "productivity difference matters most when two programmers are writing code… [it is] irrelevant during those times when both are, for example, trapped in an unnecessary meeting" (Cohn and Ford, 2002). When fully engaged, a team will move quickly. If there are too many slow people, the whole team will slow down or move forward without them.

One team was overly zealous and did not anticipate productivity decrease during transition and did not use forethought well enough. The conclusion is that "this team did not have the discipline required for XP and, while paying lip service to XP, they were actually doing nothing more than hacking" (Cohn and Ford, 2002).

The authors' experience is that testers are even more prone to view Agile as micromanagement. In typical organizations, testers do not receive much attention from managers and are not used to the extra attention they get in Agile processes. Involving testers in the daily routine as soon as possible poses one solution, but they should not write code or unit tests for programmers.

A common experience is that managers are reluctant to give up the feeling of control they get from documents typically are generated by document-driven methodologies. The solution is to show where past commitments have been incorrect (time/date/cost/functionality), so that management can be convinced to try Agile Methods.

A surprising experience is that the Human Resource (HR) department can be involved in a project adopting Agile processes. The authors experienced several cases where HR received complaints by developers who did not like the process. For example, they

received specific complaints regarding pair programming. Working with and informing HR beforehand so that they are prepared to deal with issues that might appear, can prevent this situation.

3.3.6 Lessons in Agility from Internet-Based Development

Scott Ambler describes two different approaches for developing software in two successful Internet startups that provided insights to what later became Agile Modeling (Ambler, 2002c).

The two companies were growing and needed to redesign their systems. They wanted to use an accredited software development process like Rational Unified Process (RUP) to gain the trust of investors while at the same time they wanted a process that would not impose a lot of bureaucracy that might slow them down. In both organizations, management and some members of the development team wanted more modeling; others thought it was a waste of time. Ambler calls the companies XYZ and PQR.

XYZ used an approach of modeling in teams. The team would design by whiteboarding. In the beginning, they were uncomfortable with whiteboarding and tried to use CASE tools instead, but they later discovered that whiteboarding was more efficient because a modeling language did not limit them and they could more quickly express their ideas.

PQR decided to hire a chief architect. The architect talked to members of the team, and designed privately. Later he published his results on the web and members of the team gave him feedback.

Both organizations developed in a highly interactive manner and released incrementally in short cycles. Both generated documentation in HTML and learned that design and documentation are separate activities. XYZ's architecture was developed more quickly, since lots of people worked in parallel. XYZ's architecture found greater acceptance since the development team participated in the architectural team. PQR's approach led to lower costs, since the chief architect worked alone. The chief architect also provided a single source control that sometimes caused a bottleneck in the process. Both approaches resulted in scalable architecture that met the needs of the organization, and both approaches worked well within a RUP environment.

Ambler shares the lessons learned from these approaches:

- People matter and were key to the success, in accordance with the Agile Manifesto: "value of individuals over processes and tools" (Beck et al., 2001).
- You do not need as much documentation as you think. Both organizations created only documentation that was useful and needed.
- Communication is critical. Less documentation led to greater communication.
- Modeling tools are not as useful as you think. The organizations tried to use UML modeling tools, but the tools generated more documentation than needed and were limited to the UML language. White boards and flipcharts, on the other hand, were very useful.

- You need a variety of modeling techniques in your toolkit. Since UML was not sufficient, both companies needed to perform process-, user interface- and data-modeling.
- Big up-front design is not required. Both organizations quickly began work without waiting months for detailed modeling and documentation before they started.
- Reuse the wheel, do not reinvent it. At XYZ, they took advantage of open source whenever possible.

3.3.7 Agile Modeling and the Unified Process

Ambler presents a case study of the introduction of a combination of Agile Modeling and Rational Unified Process (Ambler, 2001b). The method was introduced in a small project. Failure would be noticeable, but would not jeopardize the whole organization. Ambler points out the importance of the organization's will to change in the success of the introduction.

Different people on the team with different backgrounds had various reactions to the method. For example, one member of the team was used to Big Design Up Front and had a hard time doing an incremental design and development. Others felt more comfortable. Management was involved and interested in the effort and satisfied to see constant progress in the project.

While whiteboards made sense to the team members, they were not comfortable with index cards and post it notes. They needed a document using the appropriate tools (Together/J, Microsoft Visio, etc.). In Ambler's opinion, the team produced too much documentation. This is, however, not necessarily negative since documenting increased their comfort level during the transition.

3.4 Other Empirical Studies

In this section we discuss a selected set of experiments and surveys on Agile Methods.

3.4.1 XP in a B2B start-up

In the paper "Extreme adoption experiences of a B2B start-up" (Hodgetts and Phillips, 2002), the authors report from a case study in which two nearly identical projects used XP and non-XP practices. The XP-project delivered the same amount of functionality during a shorter period of time and required considerably less effort than the non-XP project. The XP project also increased code quality with test-first. Resulting in a 70% reduction in bugs and increased architectural quality. The value of the study is questionable, however, as the non-XP project was stopped 20 months into the project "because of excessive costs of ownership" *(Hodgetts and Phillips, 2002)* and the XP project "was suspended after nine months of development" *(Hodgetts and Phillips, 2002).* The conclusions are thus based on extrapolations of the unfinished projects and not on complete projects.

3.4.2 Empirical Experiments with XP

In order to compare XP and traditional methodologies, the authors ran a pilot XP experiment (Macias et al., 2002). The study involved eighty 2nd year undergraduate students as part of a project for real clients. The students were divided into fifteen teams

working for three clients. During five weeks, each of the three clients described what their software needs were. After that, the software was developed. Some teams used XP while others did not. At the end of the semester, five versions of the system that the each of the clients had specified were produced. The clients evaluated the quality of the systems without knowing which systems were developed using XP and which ones were not.

This experiment demonstrated that the XP teams generated more useful documentation and better manuals than the other teams. Two of the three clients found that the best external factors were in the products produced by the XP teams. The lecturers concluded that the products delivered by the XP teams possessed better internal qualities.

3.4.3 Survey conducted by Cutter Consortium

Cockburn and Highsmith mention results from a survey conducted by the Cutter Consortium in 2001. Two hundred people from organizations from all over the world responded the survey (Cockburn and Highsmith, 2001b). The findings pointed out by Cockburn and Highsmith are:

- Compared to a similar study in 2000, many more organizations were using at least one Agile Method
- In terms of business performance, customer satisfaction and quality, Agile Methods showed slightly better results than traditional methods.
- Agile Methods lead to better employee morale.

3.4.4 Quantitative Survey on XP Projects

Rumpe and Schröder report the results of a survey conducted in 2001 (Rumpe and Schröder, 2002). Forty-five participants involved in XP projects from companies of various sizes and different international locations completed the survey. Respondents had different levels of experience and participated in finished and in-progress projects using XP.

The main results of the survey indicate that most projects were successful and all of the developers would use XP on the next project if appropriate. The results also indicate that most problems are related to resistance to change: developers refused to do pair programming and managers were skeptical, etc. Common code ownership, testing and continuous integration were the most useful practices. Less used and most difficult to apply were metaphor and on-site customer. The success factors most often mentioned were testing, pair programming and the focus of XP on the right goals.

The authors point out potential problems with the survey. XP might be deemed successful due to the fact that the respondents were happy with XP. Others that had bad experiences with XP might not have been reached or did not answer. Also, early adopters tend to be highly motivated, which may be responsible for projects' success.

Interestingly, the results showed that there are larger projects that use XP. From the total of responses:

- 35.6% teams had up to 5 people,
- 48.9% teams had up to 10 people,

- 11.1% teams had up to 15 people, and,
- 4.4% teams had up to 40 people.

The survey asked respondents to rate project progress and results relative to traditional approaches in a scale from 5 (much better) to –5 (much worse). On average, respondents rated the cost of late changes, the quality of results, the fun factor of work, and on-time delivery higher than a three on this scale. No negative ratings were given. The authors divided the results between the finished and ongoing projects. It is interesting to note that both the cost of change and quality were deemed less positive by the finished projects than the ongoing ones. The authors suggest that this sustains the fact that changes in later phases still have higher costs.

3.4.5 How to Get the Most out of XP and Agile Methods

Reifer reports the results of a survey of thirty-one projects that used XP/Agile Methods practices (Reifer, 2002). The goals of the survey were to identify the practices being used, their scope and conditions, the costs and benefits of their use and lessons learned.

Most projects were characterized by small teams (less than ten participants), with the exception of one project that had thirty engineers. All projects were low-risk and lasted one-year or less. The primary reason for applying XP/Agile Methods was to decrease time-to-market.

Startup seemed most difficult for the majority of the organizations: Enthusiastic staff that wanted to try new techniques needed to convince management. Practices introduced in pilot projects represented low-risk to the organization.

The projects noticed an average gain of 15% - 23% in productivity, 5% - 7% cost reduction on average and 25% - 50% reduction in time to market.

The paper also points out 4 success factors:

- Proper domain fit: XP/Agile Methods have been recognized as working best on small projects, where systems being developed are precedent, requirements are stable and architecture is well established.
- Suitable state of organizational readiness: XP/Agile requires a cultural change. Make sure the workforce is well trained and educated.
- Process focus: Adapt and refine instead of throwing away what you have. Agile projects work best when integrated into an existing framework.
- Appropriate practice set: Do not be afraid to put new practices in place when they are needed to get the job done.

3.4.6 Costs and Benefits of Pair Programming

Pair Programming, one of the key practices of XP, marks a radical departure from traditional methods and has been the focus of some controversy. Pair programming has been argued to improve quality of software and improve successes of projects by increasing communication in the teams. Others, however, are skeptical because it seems to take two people to do the work of one, and some developers do not feel comfortable working in pairs. Pros and cons as well as main concepts, best practices, and practical

advice to successfully apply Pair Programming, are discussed in a paper based on an experiment at the University of Utah where one third of the class developed the projects individually and the rest developed in pairs. The results were analyzed from the point of views of economics, satisfaction, and design quality (Cockburn and Williams, 2000).

- Economics: The results showed that the pairs only spent 15% more time to program than the individuals and the code produced by pairs had 15% fewer defects. Thus, pair programming can be justified purely on economic grounds since the cost of fixing defects is high (Cockburn and Williams, 2000).

- Satisfaction: Results from interviews with individuals who tried pair programming were analyzed. Although some were skeptical and did not feel comfortable at first, most programmers enjoyed the experience (Cockburn and Williams, 2000).

- Design quality: In the Utah study, the pairs not only completed their projects with better quality but also implemented the same functionality in fewer lines of code. This is an indication of better design (Cockburn and Williams, 2000).

Other benefits of pair programming are continuous reviews, problem solving, learning and staff and project management (Cockburn and Williams, 2000).

- Continuous reviews: Pair programming serves as a continual design and code review that helps the removal of defects.
- Problem solving: The teams found that developing in pairs they had the ability to solve problems faster
- Learning: The teams emphasized how much they learned from each other by doing pair programming. Pair programmers often mention that they also learned to discuss and work together improving team communications and effectiveness.
- Staff and project management: From the staff and project management point of view, since people are familiar with each piece of code, staff-loss risks are reduced.

Pair programming is further discussed in a new book by Williams and Kessler (Williams and Kessler, 2003).

4 Conclusions

Agile Methods are here to stay, no doubt about it. Agile Methods will probably not "win" over traditional methods but live in symbiosis with them. While many Agile proponents see a gap between Agile and traditional methods, many practitioners believe this narrow gap can be bridged. Glass even thinks that "[t]raditionalists have a lot to learn from the Agile folks" and that "traditional software engineering can be enriched by paying attention to new ideas springing up from the field" (Glass, 2001).

Why will Agile Methods not out rule traditional methods?
Agile Methods will not out rule traditional methods because diverse processes for software engineering are still needed. Developing software for a space shuttle is not the same as developing software for a toaster (Lindvall and Rus, 2000). Not to mention the need to maintain software, typically a much bigger concern than development, that also differs according to the circumstances (Rus et al., 2002). Software maintenance is, however, not an issue discussed in Agile circles yet, probably because it is too early to draw any conclusions on how Agile Methods might impact software maintenance.

So what is it that governs what method to use?
One important factor when selecting a development method is the number of people involved, i.e. project size. The more people involved in the project, the more rigorous communication mechanisms need to be. According to Alistair Cockburn, there is one method for each project size, starting with Crystal Clear for small projects and as the project grows larger, the less Agile the methods become (Cockburn, 2000).

Other factors that have an impact on the rigor of the development methods' are application domain, criticality, and innovativeness (Glass, 2001). Applications that may endanger human life, like manned space missions, must, for example, undergo much stricter quality control than less critical applications. At the same time, a traditional method might kill projects that need to be highly innovative and are extremely sensitive to changes in market needs.

In conclusion, the selection of a method for a specific project must be very careful, taking into consideration many different factors including those mentioned above. In many cases, being both Agile and stable at the same time will be necessary. A contradictory combination, it seems, and therefore extra challenging, but not impossible: As Siemens states, "We firmly believe that agility is necessary, but that it should be built on top of an appropriately mature process foundation not instead of it" (Paulisch and Völker, 2002).

Where is Agile going?
Agile is currently an umbrella concept encompassing many different methods. XP is the most well known Agile Method. While there may always be many small methods due to the fact that their proponents are consultants who need a method to guide their work, we expect to see some consolidation in the near future. We compare the situation to events in the object-oriented world in the 1990s where many different gurus promoted their own methodology. In a few years, Rational, with Grady Booch, became the main player on the

method market by recruiting two of the main gurus: James Rumbaugh (OMT) and Ivar Jacobsson (Objectory). Quickly the "three amigos" abandoned the endless debates regarding whose method was superior, which mainly came down to whether objects are best depicted as clouds (Booch), rectangles (OMT), or circles (Objectory), and instead formed a unified alliance to quickly become the undisputed market leader for object-oriented methods. We speculate that the same can happen to the Agile Methods, based, for example, on the market-leader XP. Even if the Agile consolidation is slow or non-existent, what most likely will happen, independent of debates defining what is and is not Agile, practitioners will select and apply the most beneficial Agile practices. They will do so simply because Agile has proven that there is much to gain from using their approaches and because of the need of the software industry to deliver better software, faster and cheaper.

5 References

[1] Abrahamsson, Pekka, Salo, Outi, Ronkainen, Jussi, and Warsta, Juhani, "Agile software development methods," VTT Publications 478, 2002.

[2] Ambler, S. W., *Agile Modeling,* John Wiley and Sons, 2002a.

[3] Ambler, S. W., "Introduction to Agile Modeling (AM)," 2002b. Available: http://www.ronin-intl.com/publications/agileModeling.pdf.

[4] Ambler, S. Agile Documentation. http://www.agilemodeling.com/essays/agileDocumentation.htm . 2001a. 12-4-2002a.

[5] Ambler, S. Agile Modeling and the Unified Process. http://www.agilemodeling.com/essays/agileModelingRUP.htm . 2001b. 12-4-2002b.

[6] Ambler, S., "Lessons in Agility from Internet-Based Development," *IEEE Software*, vol. 19, no. 2, pp. 66-73, Mar. 2002c.

[7] Ambler, S. When Does(n't) Agile Modeling Make Sense? http://www.agilemodeling.com/essays/whenDoesAMWork.htm . 2002d. 12-4-2002d.

[8] Bailey, Peter, Ashworth, Neil, and Wallace, Nathan, "Challenges for Stakeholders in Adopting XP," in *Proc. 3rd International Conference on eXtreme Programming and Agile Processes in Software Engineering - XP2002*, 2002, 86-89. Available: http://www.xp2002.org/atti/Bailey-Ashworth--ChallengesforStakeholdersinAdoptingXP.pdf.

[9] Basili, Victor R., Tesoriero, Roseanne, Costa, Patricia, Lindvall, Mikael, Rus, Ioana, Shull, Forrest, and Zelkowitz, Marvin V., "Building an Experience Base for Software Engineering: A report on the first CeBASE eWorkshop," in *Proc. Profes (Product Focused Software Process Improvement)*, 2001, 110-125. Available: http://citeseer.nj.nec.com/basili01building.html.

[10] Beck, K., "Embrace Change with Extreme Programming," *IEEE Computer*, pp. 70-77, Oct. 1999a.

[11] Beck, K., *Extreme Programming Explained: Embracing Change,* Addison-Wesley, 1999b.

[12] Beck, K., Cockburn, A., Jeffries, R., and Highsmith, J. Agile Manifesto. http://www.agilemanifesto.org . 2001. 12-4-2002.

[13] Boehm, B., "A Spiral Model of Software Development and Enhancement," *IEEE Computer*, vol. 21, no. 5, pp. 61-72, 1988.

[14] Boehm, B., "Get Ready for Agile Methods, With Care," *IEEE Computer*, pp. 64-69, Jan. 2002.

[15] Bowers, P., "Highpoints From the Agile Software Development Forum," *Crosstalk*, pp. 26-27, Oct. 2002.

[16] Coad, P., deLuca, J., and Lefebvre, E., *Java Modeling in Color with UML,* Prentice Hall, 1999.

[17] Cockburn, A., "Selecting a project's methodology," *IEEE Software*, vol. 17, no. 4, pp. 64-71, 2000.

[18] Cockburn, A., "Agile Software Development Joins the "Would-Be" Crowd," *Cutter IT Journal*, pp. 6-12, Jan. 2002.

[19] Cockburn, A. and Highsmith, J., "Agile Software Development: The Business of Innovation," *IEEE Computer*, pp. 120-122, Sept. 2001a.

[20] Cockburn, A. and Highsmith, J., "Agile Software Development: The People Factor," *IEEE Computer*, pp. 131-133, Nov. 2001b.

[21] Cockburn, A. and Williams, L., "The Costs and Benefits of Pair Programming," in *Proc. eXtreme Programming and Flexible Processes in Software Engineering - XP2000* , 2000. Available: http://collaboration.csc.ncsu.edu/laurie/Papers/XPSardinia.PDF.

[22] Cohn, M. and Ford, D. Introducing an Agile Process to an Organization. http://www.mountaingoatsoftware.com/articles/IntroducingAnAgileProcess.pdf . 2002. 8-2-2002.

[23] Deias, Roberto, Giampiero Mugheddu, and Murru, Orlando, "Introducing XP in a Start-Up," in *Proc. 3rd International Conference on eXtreme Programming and Agile Processes in Software Engineering - XP2002*, 2002, 62-65. Available: http://www.xp2002.org/atti/Deias-Mugheddu--IntroducingXPinastart-up.pdf.

[24] DeMarco, T. and Boehm, B., "The Agile Methods Fray," *IEEE Computer*, pp. 90-92, June 2002.

[25] Elssamadisy, Amr and Schalliol, Gregory, "Recognizing and Responding to "Bad Smells" in Extreme Programming," 2002, 617-622.

[26] Glass, R., "Agile Versus Traditional: Make Love, Not War," *Cutter IT Journal*, pp. 12-18, Dec. 2001.

[27] Glazer, H., "Dispelling the Process Myth: Having a Process Does Not Mean Sacrificing Agility or Creativity," *Crosstalk*, Nov. 2001.

[28] Grenning, J., "Launching Extreme Programming at a Process-Intensive Company," *IEEE Software*, vol. 18, no. 6, pp. 27-33, Nov. 2001.

[29] Highsmith, J., *Agile Software Development Ecosystems,* Boston, MA: Addison-Wesley, 2002a.

[30] Highsmith, J., "What Is Agile Software Development?," *Crosstalk*, pp. 4-9, Oct. 2002b.

[31] Highsmith, J. and Cockburn, A., "Agile Software Development: The Business of Innovation," *IEEE Computer*, pp. 120-122, Sept. 2001.

[32] Highsmith, J., Orr, K., and Cockburn, A., "Extreme Programming," *E-Business Application Delivery*, pp. 4-17, Feb. 2000. Available: http://www.cutter.com/freestuff/ead0002.pdf.

[33] Hodgetts, P. and Phillips, D. Extreme Adoption Experiences of a B2B Start-up. http://www.extremejava.com/eXtremeAdoptioneXperiencesofaB2BStartUp.pdf . 2002. 12-4-2002.

[34] Humphrey, W. S., *A Discipline for Software Engineering,* Reading, MA: Addison Wesley Longman, Inc., 1995.

[35] Jeffries, R. Extreme Programming and the Capability Maturity Model. http://www.xprogramming.com/xpmag/xp_and_cmm.htm . 2000. 12-4-2002.

[36] Karlström, Daniel, "Introducing Extreme Programming – An Experience Report," in *Proc. 3rd International Conference on eXtreme Programming and Agile Processes in Software Engineering - XP2002*, 2002, 24-29. Available: http://www.xp2002.org/atti/DanielKarlstrom--IntroducingExtremeProgramming.pdf.

[37] Lindvall, Mikael, Basili, Victor R., Boehm, Barry, Costa, Patricia, Dangle, Kathleen, Shull, Forrest, Tesoriero, Roseanne, Williams, Laurie, and Zelkowitz, Marvin V., "Empirical Findings in Agile Methods," in *Proc. Extreme Programming and Agile Methods - XP/Agile Universe 2002*, 2002a, 197-207. Available: http://fc-md.umd.edu/mikli/Lindvall_agile_universe_eworkshop.pdf.

[38] Lindvall, Mikael, Basili, Victor R., Boehm, Barry, Costa, Patricia, Shull, Forrest, Tesoriero, Roseanne, Williams, Laurie, and Zelkowitz, Marvin V., "Results from the 2nd eWorkshop on Agile Methods," Fraunhofer Center for Experimental Software Engineering, College Park, Maryland 20742, Tech. Rep. Technical Report 02-109, Aug., 2002b.

[39] Lindvall, M. and Rus, I., "Process Diversity in Software Development," *IEEE Software*, vol. 17, no. 4, pp. 14-71, Aug. 2000. Available: http://fc-md.umd.edu/mikli/LindvallProcessDiversity.pdf.

[40] Macias, Francisco, Holcombe, Mike, and Gheorghe, Marian, "Empirical experiments with XP," in *Proc. 3rd International Conference on eXtreme Programming and Agile Processes in Software Engineering - XP2002*, 2002, 225-228. Available: http://www.xp2002.org/atti/Macias-Holcombe--EmpiricalexperimentswithXP.pdf.

[41] Paulisch, Frances and Völker, Axel, "Agility - Build on a Mature Foundation," in *Proc. Software Engineering Process Group Conference - SEPG 2002*, 2002.

[42] Paulk, M. C., "Extreme Programming from a CMM Perspective," *IEEE Software*, vol. 18, no. 6, pp. 19-26, 2001.

[43] Paulk, M. C., "Agile Methodologies and Process Discipline," *Crosstalk*, pp. 15-18, Oct. 2002.

[44] Paulk, Mark C., "Key Practices of the Capability Maturity Model, Version 1.1," Technical Report, Tech. Rep. CMU/SEI-93-TR-25, 1993.

[45] Poole, C. and Huisman, J., "Using Extreme Programming in a Maintenance Environment," *IEEE Software*, vol. 18, no. 6, pp. 42-50, Nov. 2001.

[46] Poppendieck, M. Lean Programming. http://www.agilealliance.org/articles/articles/LeanProgramming.htm . 2001. 4-12-2002.

[47] Puttman, David, "Where Has All the Management Gone? " in *Proc. 3rd International Conference on eXtreme Programming and Agile Processes in Software Engineering - XP2002*, 2002, 39-42. Available: http://www.xp2002.org/atti/DavidPutman--WhereHasAllTheManagementGone.pdf.

[48] Rakitin, S. R., "Manifesto Elicits Cynicism," *IEEE Computer*, vol. 34, no. 12, pp. 4, Dec. 2001.

[49] Reifer, D., "How the Get the Most out of Extreme Programming/Agile Methods," in *Proc. Extreme Programming and Agile Methods - XP/Agile Universe 2002*, 2002, 185-196.

[50] Royce, W. W., "Managing the Development of Large Software Systems: Concepts and Techniques," in *Proc. WESCON*, 1970, 1-9.

[51] Rumpe, Bernhard and Schröder, A., "Quantitative Survey on Extremme Programming Project," 2002. Available: http://www.xp2002.org/atti/Rumpe-Schroder--QuantitativeSurveyonExtremeProgrammingProjects.pdf.

[52] Rus, I., Seaman, C., and Lindvall, M., "Process Diversity in Software Maintenance - Guest editors' introduction (Accepted for publication)," *Software Maintenance Research and Practice*, Dec. 2002.

[53] Schwaber, K. and Beedle, M., *Agile Software Development with SCRUM,* Prentice-Hall, 2002.

[54] Schwaber, K. Controlled Chaos: Living on the Edge. http://www.agilealliance.org/articles/articles/ap.pdf . 2002. 4-12-2002.

[55] Shull, Forrest, Basili, Victor R., Boehm, Barry, Brown, A. W., Costa, Patricia, Lindvall, Mikael, Port, D, Rus, Ioana, Tesoriero, Roseanne, and Zelkowitz, Marvin V., "What We Have Learned About Fighting Defects," in *Proc. 8th International Software Metrics Symposium*, 2002, 249-258. Available: http://fc-md.umd.edu/fcmd/Papers/shull_defects.ps.

[56] Stapleton, J., *DSDM: The Method in Practice,* Addison Wesley Longman, 1997.

[57] The C3 Team, "Chrysler Goes to "Extremes" ," *Distributed Computing*, pp. 24-28, Oct. 1998.

[58] Turk, Dan, France, Robert, and Rumpe, Bernhard, "Limitations of Agile Software Processes," in *Proc. 3rd International Conference on eXtreme Programming and Agile Processes in Software Engineering - XP2002*, 2002. Available: http://www4.informatik.tu-muenchen.de/~rumpe/ps/XP02.Limitations.pdf.

[59] Turner, Richard and Jain, Apurva, "Agile meets CMMI: Culture clash or common cause?," in *Proc. EXtreme Programming and Agile Methods - XP/Agile Universe 2002*, 2002, 153-165.

[60] Vic Basili and Turner, A. J., "Iterative Enhancement: A Practical Technique for Software Development," *IEEE Transactions on Software Engineering*, vol. 1, no. 4, pp. 390-396, 1975.

[61] Victoria Bellotti, Burton, Richard R., Ducheneaut, Nicolas, Howard, Mark, Neuwirth, Christine, and Smith, Ian, "XP in a Research Lab: The Hunt for Strategic Value," in *Proc. 3rd International Conference on eXtreme Programming and Agile Processes in Software Engineering - XP2002*, 2002, 56-61. Available: http://www.xp2002.org/atti/Bellotti-Burton--XPInAResearchLab.pdf.

[62] Williams, L. and Kessler, R. R., *Pair Programming Illuminated,* Addison-Wesley, 2003.

[63] Williams, L., Kessler, R. R., Cunningham, W., and Jeffries, R., "Strengthening the case for pair programming," *IEEE Software*, vol. 17, no. 4, pp. 19-25, 2000.

6 Appendix: An analysis of Agile Methods

'Agile' has become a buzzword in the software industry. Many methods and processes are referred to as 'Agile': making it difficult to distinguish between one and the next. There is a lack of literature on techniques with which to compare software development methods, so we have developed processes through which to draw this comparison. This technique will not be the focus of this section, nor do we guarantee its comprehensiveness, but we found it adequate for our analysis, which we will discuss in detail below.

While conducting our research, we found it difficult to distinguish between methods in respect to which aspect of software development each method targeted. To help with our own understanding, we decided to examine each method in terms of what activities it supports, and to what extent. All methods, whether traditional or Agile, address the following project aspects to varying degrees: development support, management support, communications support, and decision-making support. Although critics may find more project areas missing from this list, these are the four we felt were most critical for Agile Methods.

The next obstacle was to find a basis for comparison between methods. For this we decided to use each method's core practices or rules. This method for comparison does have its drawbacks:

- Some methods, like XP, have a discreet collection of practices while others, like Scrum, are not as clearly delineated.
- Processes such as Lean Development (LD) present more principles than practices. LD's "Satisfying the customer is the highest priority" principle, for instance, is stated at a much more abstract level than Scrum's Constant Testing practice.
- Relying on a method's stated core practices naturally leaves a lot of the method behind. Daily standup meetings practiced in XP are not explicitly stated in the 12 practices but nonetheless emphasizes XP's attention to communication.

Despite these acknowledged limitations, we feel each method's core practices or principles provide a good representation of the method's focus. Researchers interested in pursuing further this line of analysis may want to explore how to adequately represent and compare methods and processes.

We began by breaking the support groups (development, management, communication, decision-making) into smaller subsections for easier analysis. Development was redefined as requirements collection/analysis, design, coding, testing/integration, and maintenance. Management was clarified as project management. Communication was split into developer-customer, developer-manager, and developer-developer communication. Decision-making was separated into release planning, design & development, and project management.

A survey was conducted asking five experts to classify the core practices of XP, Scrum, Lean Development, FDD, and DSDM. The results were averaged and color-coded in an attempt to create easily readable results. Support agreement of more than 60% is black, 0-59% is white, and 0% (all experts agreed there is no support) is gray. A brief explanation of well-supported (black) aspects follows each chart.

6.1 Extreme Programming

XP was the method best understood by our experts; all five responded. XP's practices are abbreviated as: The Planning Game (PG), Small Releases (SR), The Metaphor (M), Simple Design (SD), Test-First Development (TF), Refactoring (R), Pair Programming (PP), Continuous Integration (CI), Collective Code Ownership (CO), On-Site Customer (OC), 40-Hour Work Week (WW), and Open Workspace (OW).

Table 2. XP Development Support

	PG	SR	M	SD	TF	R	PP	CI	CCO	OSC	WW	OW
Requirements												
Design												
Coding												
Testing/ Integration												
Maintenance												

- PG: The Planning Game is used for Requirements collection and clarification at the beginning of each iteration and is also employed to address Maintenance issues between iterations.
- SR: Small Releases forces developers to Design in components and Test after each release.
- M: Interestingly, our experts found Metaphor, the oft-cited least understood practice, to provide support for all phases of development. By creating one or a set of metaphors to represent the system, decisions involving Requirements, Design, Coding, Testing/Integration, and Maintenance can be easily evaluated for relevance and priority.
- SD: Simple Design helps a developer choose their Design, tells them how to Code sections when presented with multiple alternatives, and makes Maintenance easier than a complicated design.
- TF: Test-First is a way of Coding as well as a Testing approach, and makes Maintenance easier by providing a test suite against which modifications can be checked.
- R: Refactoring tells a developer to simplify the Design when she sees the option, effects Coding in the same way, and facilitates Maintenance by keeping the design simple.
- PP: Pair Programming has two developers code at the same computer, and lets them collaborate with Designing and Coding. Maintenance is affected because

two developers working together will usually produce less, and better written code than a single developer working alone.

- CI: Continuous Integration is an approach to Coding, and obviously effects how and when developers Integrate new code.
- CCO: Collective Code Ownership is a way for developers to program, giving them the option to modify other's Code.
- OSC: On Site Customer impacts Requirements because developers may discuss and clarify requirements at any time.
- OW: Open Workspace allows all developers, even those beyond the pair programming team, to collaborate with Coding and Integration.

Table 3. XP Management Support

	PG	SR	M	SD	TF	R	PP	CI	CCO	OSC	WW	OW
Management	■	■		■		■	■	■	■	■	■	■

- PG: The Planning Game allows the project manager to meet with all the project stakeholders to plan the next iteration.
- SR: Small Releases tell the manager how often to iterate.
- CI: Continuous Integration allows the project manager (PM) to see the current state of the system at any point in time.
- OSC: On Site Customer enables the manager to better interact with the customer than she would be able to with an offsite customer.
- WW: The 40 hour Work Week provides a philosophy on how to manage people.
- OW: Open Workspace tells the PM how the work environment should be set up.

Table 4. XP Communication Support

	PG	SR	M	SD	TF	R	PP	CI	CCO	OSC	WW	OW
Developer-Customer	■	■	■	■		■	■		■	■	■	
Developer-Manager	■	■	■	■		■		■	■	■		■
Developer-Developer	■	■	■	■	■	■	■	■	■	■		■

- PG: The Planning Game helps the Developers communicate with the Customer, the Manager, and other Developers, in the beginning of each iteration.
- SR: Small Releases provide instant project progress assessment for Customers, Managers, and Developers between iterations.
- M: Using a Metaphor or a set of metaphors allows Customers, Managers, and Developers to communicate in a common, non-technical language.
- SD: Simple Design encourages Developers to communicate their ideas as simply as possible.
- TF: Test-First allows Developers to communicate the purpose of code before it is even developed.

- R: Refactoring encourages Developers to simplify code, making the design simpler and easier to understand.
- PP: Pair Programming allow sets of Developers to communicate intensely while coding.
- CI: Continuous Integration allows the Managers and Developers to check the current state of the system at any time.
- CCO: Collective Code Ownership allows Developers to communicate through code, comments, and documentation.
- OSC: On Site Customer facilitates quick communication between the Customer and the Developers.
- OW: Open Workspace enables Developers and Managers to communicate quickly and freely.

Table 5. XP Decision-Making Support

	PG	SR	M	SD	TF	R	PP	CI	CCO	OSC	WW	OW
Release Planning	■	■		■		■	■		■			
Design and Development			■	■	■		■	■	■			
Project Management	■	■		■		■			■	■		

- PG: The Planning Game assists decision making for Releases and helps Project Panagers Plan the project.
- SR: Small Releases dictates how often to iterate which affects Release Planning and Project Management.
- M: Metaphor guides Design decisions based on how well the design fits the metaphor.
- SD: Simple Design guides Design decisions when presented with multiple choices.
- TF: Test-First tells the developer that before Designs and Develops any new code he must first write the test.
- PP: Pair Programming lets programmers collaborate on Design and Development decisions.
- CI: Continuous Integration instructs the programmers to integrate on a regular basis, which affects how Design and Development is conducted.
- CCO: Collective Code Ownership encourages developers to make changes to parts of the code that they did not author instead of waiting for the original developer to get around to it and affects how Design and Development is conducted.
- OSC: On Site Customer allows the customer and PM to interact frequently, enabling quick decision-making.

6.2 Scrum

Only three experts felt comfortable answering about Scrum. The core practices are abbreviated: Small Teams (ST), Frequent Builds (FB), Low-Coupling Packets (LCP), Constant Testing (CT), Constant Documentation (CD), Iterative Controls (IC), Ability to declare the project one at any time (DPD).

Table 6. Scrum Development Support

	ST	FB	LCP	CT	CD	IC	DPD
Requirements							
Design							
Coding							
Testing/ Integration							
Maintenance							

- ST: Breaking the development group into Small Teams affects how the system is Designed and distributed between teams.
- FB: Frequent Builds affects Coding practices and means that new code needs to be Integrated on a regular basis. Maintenance is also affected, as a current version of the system is always available for testing, catching errors earlier.
- LCP: Low-Coupling Packets influences the system Design and Coding practices. Testing, Integration, and Maintenance should be made easier due to relative component independence.
- CT: Constant Testing changes the way developers need to Code, Test and Integrate, and should make Maintenance easier by catching more bugs during development.
- CD: Constant Documentation affects the way Requirements, Design, and Coding are conducted. The presence of up-to-date documentation should facilitate testing and maintenance.
- IC: Iterative Controls help prioritize and guide Requirements collection and Design. They also affect how Coding, and Testing and Integration are conducted.
- DPD: The ability to declare a project done at any time has far reaching consequences; every step in the development process should be treated as if it were in the last iteration.

Table 7. Scrum Management Support

	ST	FB	LCP	CT	CD	IC	DPD
Management							

- ST: Small Teams means Managers have to manage and distribute work between teams and team leaders.

- FB: Frequent Builds allows Managers to see the state of the system at any given time to track progress.
- LCP: Low Coupling Packets influences how Managers distributed work.
- CT: Constant Testing provides the Manager with a system he can demo or ship at any time.
- CD: Constant Documentation provides an up to date snapshot of the system and its progress, which can be used by the Manager for tracking or for bringing new people up to speed.
- IC: Iterative Controls help the Manager gauge requirements, functionality, risk, and plan iterations.
- DPD: The ability to declare a product done at any time is a Management philosophy placing emphasis on usability and correctness of the system rather than strict feature growth.

Table 8. Scrum Communication Support

	ST	FB	LCP	CT	CD	IC	DPD
Developer-Customer							
Developer-Manager							
Developer-Developer							

- ST: Small Teams help break down communications barriers allowing easy, informal communication between all parties in the teams.
- FB: Frequent Builds enables Developers to communicate the status of the system with other Developers and Managers at any time.
- LCP: Low Coupling Packets reduce the need for technical communications between Developers.
- CT: Constant Testing allows Developers to know the current state of the system at any point in time.
- CD: By providing comprehensive up-to-date documentation, any stakeholder can learn about their respective interests in the system.
- IC: Iterative Controls provide a means through which Customers, Management, and Developers collaborate to plan iterations.

Table 9. Scrum Decision Making Support

	ST	FB	LCP	CT	CD	IC	DPD
Release Planning							
Design and Development							
Project Management							

- ST: Small Teams make all levels of decision-making easier by involving a smaller number of individuals on lower level decisions.
- FB: Frequent Builds help Managers plan and monitor Releases and Development.
- LCP: Low-Coupling Packets helps guide Design decisions.
- CT: Constant Testing tells developers to test as they Code.
- CD: Constant Documentation dictates that documentation should be produced and kept up to date not only for code but also for Requirements and Release Planning. The produced documentation helps guide the PM.
- IC: Iterative Controls help guide the PM with respect to Release Planning decisions.
- DPD: The ability to declare a project done at any time effects what kind of features or fixes are incorporated into the next Release, and also effects the mentality with which the PM Manages the Project.

6.3 Lean Development

The 12 principles of LD are abbreviated as: Satisfying the Customer is the highest priority (SC), always provide the Best Value for the money (BV), success depends on active Customer Participation (CP), every LD project is a Team Effort (TE), Everything is Changeable (EC), Domain not point Solutions (DS), Complete do not construct (C), an 80 percent solution today instead of 100 percent solution tomorrow (80%), Minimalism is Essential (ME), Needs Determine Technology (NDT), product growth is Feature Growth not size growth (FG), and Never Push LD beyond its limits (NP). 3 experts contributed to the LD survey.

Table 10. Lean Development Development Support

	SC	BV	CP	TE	EC	DS	C	80%	ME	NDT	FG	NP
Requirements												
Design												
Coding												
Testing/ Integration												
Maintenance												

- BV: Always provide the Best Value for the money means that during requirements analysis, easy to implement features that provide a quick win rather than hard to implement features that do not provide immediate value. Similarly, this effects Coding and Design: they should be done with optimal trade off between quality and time.
- CP: Requirements collection and analysis works best with active Customer Participation.
- TE: Every phase of development is a Team Effort.

- DS: By focusing on Domain Solutions, Design and Coding should look to create reusable components. Domain solutions will be pre-tested and should be easier to Integrate and Maintain than brand new code.
- C: When Designing to Construct a new system, LD teams look to purchase parts of the system that may already be commercially available. By doing so, Testing and Integration should be easier, as the shrink-wrapped portion is, ideally, bug-free.

Table 11. Lean Development Management Support

	SC	BV	CP	TE	EC	DS	C	80%	ME	NDT	FG	NP
Management	■	■	■	■				■	■			

- SC: The PM needs to change her frame of mind to make Customer Satisfaction the highest priority, as opposed to budget, politics, and other concerns.
- BV: The PM also needs to manage the project with the goal to build and prioritize the system to provide the Best Value for the money.
- CP: It becomes the PM's responsibility to keep the Customer Participating in the project.
- TE: The PM needs to include the entire Team in decision-making processes.
- 80%: Instead of making everything perfect, the PM should focus on providing the best system she can at the moment.
- ME: The PM should focus on keeping team size, code size, documentation, and budget as small as is necessary for a successful project.

Table 12. Lean Development Communications Support

	SC	BV	CP	TE	EC	DS	C	80%	ME	NDT	FG	NP
Developer-Customer	■		■	■			■	■	■	■	■	■
Developer-Manager			■	■	■		■	■		■	■	■
Developer-Developer	■	■	■	■			■	■	■	■	■	■

- SC: Ensuring Customer Satisfaction entails enhanced communication between the Developers and Customers.
- CP: Active Customer Participation gives Customers more incentive to work with the Developers.
- TE: The 'everything is a Team Effort' philosophy encourages communication between all members of the team.

Table 13. Lean Development Decision Making Support

	SC	BV	CP	TE	EC	DS	C	80%	ME	NDT	FG	NP
Release Planning	■	■	■	■	■	■	■	■		■	■	■
Design and Development	■		■		■		■	■		■		■
Project Management	■	■	■	■			■	■	■	■	■	■

- SC: Prioritizing Customer Satisfaction means that during Release Planning, Design and Development, and Project Management, the interest of the customer may have to be put before that of the team.
- BV: Providing the Best Value for the money is a management philosophy, affecting mostly what requirements get prioritized for what release.
- CP: Active Customer Participation provides decision support for PM's, and is also instrumental in prioritizing Release features.
- EC: Having the ability to Change Everything means that Release and Design decisions are not set in stone, letting them be made more quickly and changed later if necessary.
- C: An emphasis Construing based on already built components has a large effect on Design decisions.
- 80%: Having an 80% solution today means that, from a Release, Design, and PM perspective, adding a new feature today is a better decision than completing an old one.
- ME: Minimalism helps a PM decide what artifacts to produce during development.
- NDT: The Needs Determine Technology philosophy helps the PM and designers decide on an appropriate solution rather than a high-tech solution for high-tech's sake.
- FG: By emphasizing Feature Growth, Releases and PM's tend to push features more than other requirements.

6.4 Feature Driven Development

The core practices of FDD are abbreviated: problem domain Object Modeling (OM), Feature Development (FD), Component/class Ownership (CO), Feature Teams (FT), Inspections (I), Configuration Management (CM), Regular Builds (RB), and Visibility of progress and results (V). Only 2 experts felt comfortable enough with FDD to complete the survey.

Table 14. Feature Driven Development Development Support

	OM	FD	CO	FT	I	CM	RB	V
Requirements								
Design								
Coding								
Testing/ Integration								
Maintenance								

- OM: Object Modeling provides a different approach to Design.
- FD: Feature Development provides a development methodology that effects the way Design, Coding, and Integration are approached. Maintenance is also affected as the system is considered as a collection of features rather than lines of code.
- CO: Individual Code Ownership means that Design, Coding, and Integration become individual efforts.
- FT: Feature Teams means that the feature as a whole becomes a team effort.
- I: Inspections are a testing technique that should produce better and more bug-free code that is easier to Maintain.
- CM: Configuration Management is established for support of Testing, Integration, and Maintenance.
- RB: Regular Builds affect coding procedures, helps to integrate testing and integration during the development process, and make maintenance easier with more bug-free code.

Table 15. Feature Driven Development Management Support

	OM	FD	CO	FT	I	CM	RB	V
Management								

- FD: Feature Development allows the PM to manage teams by easily separating development workload.
- CO: Code Ownership gives the PM a point of contact about any piece of the system.
- FT: Feature Teams allow the PM to break the team effort into more easily manageable sections.
- RB: Regular Builds gives the PM a snapshot of the system at any point in time.
- V: Visibility of progress allows easy tracking of the project.

Table 16. Feature Driven Development Communication Support

	OM	FD	CO	FT	I	CM	RB	V
Developer-Customer								
Developer-Manager								
Developer-Developer								

- OM: Object Modeling allows Developers to communicate with Managers and other Developers specifically and in detail about small components of the system.
- FD: Feature Development allows the Developer to prototype and display working units of the system to Managers and Customers.
- CO: Code Ownership gives Managers and other Developers a point of contact about specific sections of code in the system.
- FT: Feature Teams allow easy collaboration and communication between Developers and Managers.
- I: Inspections allow Developers to read, explain, and understand the code.
- CM: Configuration Management provides a vehicle for communication for Developers and Managers.
- RB: Regular Builds let Developers and Managers see the current state of the system.
- V: Progress Visibility allows the Customer to track the project with ease.

Table 17. Feature Driven Development Decision Making Support

	OM	FD	CO	FT	I	CM	RB	V
Release Planning								
Design and Development								
Project Management								

- OM: Object Modeling allows for a flexible framework for Design.
- FD: Feature Development allows for easy distribution of features in releases. Prototyped features can be tested, designed, and developed. And Project Managers can manage the system as a set of features.
- CO: Code Ownership gives PM's a point of contact for specific pieces of code.
- FT: By building small team to handle Features, decision making for release and design and development is delegated to the group. It also guides PM's resource allocation.
- I: Inspections correct and reshape design and code.
- CM: Configuration Management provides a resource and reference for PM's.
- RB: Regular Builds provide feedback during Development.

- V: Visibility allows the project manager to track the project and make changes when necessary.

6.5 Dynamic Systems Development Methodology

Only one expert felt comfortable enough with DSDM to complete the survey. DSDM's principles are abbreviated: Active User Involvement is imperative (AUI), DSDM teams must be Empowered to make decisions (E), focus is on Delivery Of Products (DOP), Fitness for business purpose is the essential criterion for acceptance of deliverables (F), Iterative and incremental development is necessary to converge on an accurate business solution (I), all changes during development are Reversible (R), requirements are baselines at a High Level (HL), Testing is Integrated throughout the life cycle (TI), a Collaborative and Cooperative approach between all stakeholder is essential (CC).

Table 18. Dynamic Systems Development Methodology Development Support

	AUI	E	DoP	F	I	R	HL	TI	CC
Requirements									
Design									
Coding									
Testing/ Integration									
Maintenance									

- AUI: Active User Involvement is important for good Requirements collection.
- E: Team Empowerment allows developers to make the right decisions during Design and Coding.
- DOP: Frequent Delivery Of Products gives the customer a system they can Test while it is still under development.
- I: Incremental development affects the entire development process, breaking Requirements collection, Design, Coding, Testing, Integration, and Maintenance into short cycles.
- R: Reversible decisions means developers can feel freer to commit to decisions during Design and Coding. During Testing or Integration these decisions can be reversed if necessary.
- HL: High Level requirements keeps Requirements collection at and abstraction high enough for participation from all stakeholders.
- TI: Constant Testing and Integration allows bugs to be caught and fixed earlier in the development lifecycle.
- CC: A Collaborative approach between stakeholders will assist in accurate Requirements collection.

Table 19. Dynamic Systems Development Methodology Management Support

	AUI	E	DoP	F	I	R	HL	TI	CC
Management									

- AUI: The Project Manager needs to manage collaboration between users and the Customer and Developers.
- E: Empowering teams means Management has to be more flexible.
- DOP: Focus on the Delivery Of Products is a Management mindset.
- F: Management needs to consider Fitness for purpose over other factors.
- I: Iterative development breaks Management into smaller, more intense cycles.
- R: The project manager needs to feel free to make decisions without worrying about irReversible consequences.
- CC: Managers need to facilitate Collaboration between stakeholders.

Table 20. Dynamic Systems Development Methodology Communication Support

	AUI	E	DoP	F	I	R	HL	TI	CC
Developer-Customer									
Developer-Manager									
Developer-Developer									

- AUI: Active User Involvement ensures good communication between Developers and the Customer.
- DOP: Frequent Delivery Of Products allows Managers and Customers to keep up to date on the status of the system.
- I: Incremental development gives Developers, Managers, and Customers frequent opportunities to interact.
- HL: High Level requirements provide Developers with a vehicle for non-technical requirements communication with Managers and Customers.
- TI: Integrated Testing allows Developers and Managers to see the state of the system at any point in time.
- CC: a Collaborative approach keeps the Customer actively involved.

Table 21. Dynamic Systems Development Methodology Decision Making Support

	AUI	E	DoP	F	I	R	HL	TI	CC
Release Planning									
Design and Development									
Project Management									

- AUI: Management needs to keep users actively involved.
- E: Teams can feel free to make design and development decisions as they see fit.

- DOP: Management philosophy needs to reflect the frequent delivery of products and plan releases accordingly.
- F: Management needs to evaluate decisions on fitness for the business purpose.
- I: Iterative development makes decision-making cycles shorter and deals with smaller, more frequent decisions.
- R: Reversible decisions means that decision making does not have to be 100% complete or hold up the process until made.
- TI: Developers learn to test frequently during development.