LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

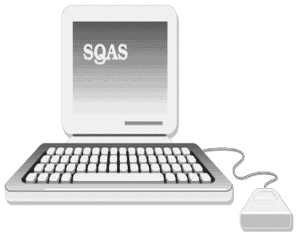
UCRL-TR-230931

Applying Agile MethodstoWeapon/Weapon-Related Software

D. Adams, M. Armendariz, M. Blackledge, F. Campbell, M. Cloninger, L. Cox, J. Davis, M. Elliott, K. Granger, S. Hans, C. Kuhn, M. Lackner, P. Loo, S. Matthews, K. Morrell, C. Owens, D. Peercy, G. Pope, R. Quirk, D. Schilling, A. Stewart, A. Tran, R. Ward, M. Williamson

May 14, 2007

This work was performed under the auspices of the U.S. Department of Energy by University of California, Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.



UCRL-TR-230931

Department of Energy Quality Managers
Software Quality Assurance Subcommittee
Reference Document SQAS31.01.00-2007

Applying Agile Methods to Weapon/Weapon-Related Software

April 2007

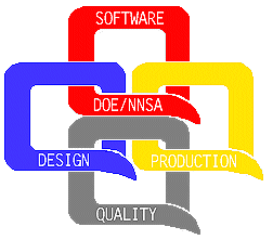
Prepared by the
Software Quality Assurance Subcommittee
of the
Nuclear Weapons Complex Quality Managers
under
United States Department of Energy
Albuquerque Operations Office
Albuquerque, New Mexico 87185

This report will be reviewed and updated on a periodic basis. If you have any corrections, additions, or deletions, please contact the Quality Manager at your site for the name of a contact on the Software Quality Assurance Subcommittee.

SQAS32.01.00-2007

Applying Agile Methods
to
Weapon/Weapon-Related Software

April 2007



United States Department of Energy

Albuquerque Operations Office

Abstract

This white paper provides information and guidance to the Department of Energy sites on software quality and software quality assurance related to applying Agile software quality engineering methods to weapon/weapon-related software development.

Acknowledgments

The Software Quality Assurance Subcommittee (SQAS) of the Nuclear Weapons Complex Quality Managers initiated Work Item #31 to develop a quality report addressing applying Agile software quality engineering methods to weapon/weapon-related software. The SQAS members who have contributed to this work item are listed below

Dennis Adams	NNSA
Maria Armendariz (Editor)	SA
Mike Blackledge	SA
Frank Campbell	SR
Mack Cloninger	SR
Larry Cox	LA
James Davis	SR
Mike Elliott	UK AWE
Ken Granger	Y12
Steve Hans	RL
Cathy Kuhn	KC
Michael Lackner	LA
Patricia Loo	INL
Scott Mathews	LA
Keith Morrell	SR
Carolyn Owens (Chair)	LL
Dave Peercy	SA
Greg Pope	LL
Robert Quirk	DNFSB
Don Schilling	KC
Ann Stewart	OR
Anton Tran	NA-173
Roger Ward	PX
Mike Williamson	SA

Table of Contents

List of Figures

List of Tables

1. Introduction

This white paper provides information and guidance to the Department of Energy (DOE) sites on Agile software development methods and the impact of their application on weapon/weapon-related software development.

1.1 Purpose and Scope

The purpose of this white paper is to provide an overview of Agile methods, examine the accepted interpretations/uses/practices of these methodologies, and discuss the applicability of Agile methods with respect to Nuclear Weapons Complex (NWC) Technical Business Practices (TBPs). It also provides recommendations on the application of Agile methods to the development of weapon/weapon-related software.

1.2 Background

Recent graduates and new hires are entering the NWC workforce environment with newly acquired emphasis on non traditional or Agile software development methods. Additionally, current members of the work force have been attending courses/conferences focusing on these new Agile software development concepts. Agile methodologies appear to be at odds with the traditional software engineering methodologies and may not provide sufficient rigor and exposure required for high integrity software. There is also the concern that Agile methods may not provide all of the necessary information and activities required for the qualification of weapon/weapon-related software. Currently no clear guidance exists within the NWC or the Technical Business Practices (TBPs) on whether or not Agile software development methods should be used in weapon/weapon-related software development.

1.3 Roadmap

An overview of Agile methods and brief examination of specific methodologies that are considered to be an Agile method are presented in Section 2. A summary of weapon/weapon-related software characteristics and requirements compared with the Agile method values and principles is presented in Section 3 along with a comparison matrix for each of the Agile methods described in Section 2. An executive summary of the barriers to using Agile methods for weapon/weapon-related software including conclusions from this white paper is presented in Section 4. Resource references, definitions, and acronyms are included in Appendix A.

2. Agile Methods

2.1 Overview

2.1.1 Background

The "agile" approach to software engineering gained popularity with the formation of the Agile Alliance in 2001 and the publication of the Agile Manifesto by a group of software practitioners and consultants.

Agile Manifesto

"We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- Individuals and interactions over processes and tools.
- Working software over comprehensive documentation.
- Customer collaboration over contract negotiation.
- Responding to change over following a plan.

That is, while there is value in the items on the right, we value the items on the left more."[1]

Agile methods for software development are described by Boehm and Turner[2] as being "very lightweight processes that employ short iterative cycles; actively involve users to establish, prioritize, and verify requirements; and rely on tacit knowledge within a team as opposed to documentation."

The reasons for this popularity, according to Boehm and Turner, are that these methods "are an outgrowth of rapid prototyping and rapid development experiences as well as the resurgence of the philosophy that programming is a craft rather than an industrial process. … The rapidly changing nature of the Internet-based economy demands flexibility and speed from software developers, something not usually associated with plan-driven development. The problem of change is exacerbated by long development cycles that yield code that may be well written but does not meet user expectations."

2.1.2 Agile Principles and Characteristics

The principles behind the Agile Manifesto[1] are:

(1) Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

(2) Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

(3) Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

(4) Business people and developers must work together daily throughout the project.

[1] Beck, K., et al., "Manifesto for Agile Software Development," Feb. 2001; **www.agilemanifesto.org**
[2] Boehm, B., Turner, R., Balancing Agility and Discipline Addison-Wesley, 2004

(5) Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

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

(7) Working software is the primary measure of progress.

(8) Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

(9) Continuous attention to technical excellence and good design enhances agility.

(10) Simplicity--the art of maximizing the amount of work not done--is essential.

(11) The best architectures, requirements, and designs emerge from self-organizing teams.

(12) At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

The main characteristics that agile methods have in common are:

- Small collocated teams,
- Collocated users/experts,
- Short, iterative, incremental development cycles,
- Experienced and knowledgeable developers.

The agile methods discussed in this section are: SCRUM, Adaptive Software Development (ASD), Lean Development (LD), Crystal, eXtreme Programming (XP), Dynamic Systems Development Method (DSDM), and Feature Driven Development (FDD). A description of each of each of these methods is provided, as well as the applicability, strengths and limitations of each method are discussed.

2.2 SCRUM

2.2.1 Description

Scrum is a lightweight process used to manage product development; it is a project management process. In the context of managing a software development product, Scrum embraces iterative and incremental practices. Scrum does not require or provide any specific software development method or practice to be used. Scrum is normally used as a project management method that wraps around existing software engineering practices. Scrums project management concepts are often used in conjunction with managing eXtreme Programming (XP) based software development practices.

Instead of being evidence driven by means of requirements documents, analysis specifications, design documents, etc., Scrum requires very few artifacts. It concentrates on managing a project and writing software that produces business value as soon as possible.

The practices and tools used in the various phases of Scrum provide a simple project management implementation that attempts to develop systems and products when requirements are rapidly changing or, due to unpredictability or complexity, are not well known. Given the

available resources, acceptable quality, and required release dates, the Scrum process uses an iterative incremental methodology that sets out to produce a potentially shippable set of functionality at the end of every iteration (usually a 30 day period). Change of project scope, technology, functionality, cost, and schedule are expected since Scrum accepts that the development process is unpredictable.

The Scrum methodology contains the main components of the Product backlog, the sprint, and the sprint backlog as its main artifacts (see Figure 2-1). The Product backlog records the requirements of the product, or at least the first pass at the requirements. The requirements do not need to be fully described because they will be fully determined during the Sprint (or iteration) that they are implemented in. The Sprint is a set of development activities conducted over a pre-defined period (usually 30 days) resulting in an executable software product. And the Sprint backlog is that set of requirements from the Product backlog that are being implemented during the Sprint.

The roles of Scrum are the Product Owner, Scrum Master, and Product Team. The Product Owner is similar to a manager or sponsor who prioritizes the Product Backlog of items (requirements) in order of importance to the project. The Scrum Master is equivalent to a Project Manager or Team Leader. They are responsible for ensuring Scrum values and practices. The Scrum Masters main job is to remove roadblocks and issues that might slow down or stop activity of the Sprint. The Project Team should consist of between 5-10 members and contain a cross-functional knowledge base, involving individuals from a multitude of disciplines: QA, Programmers, UI Designers, etc. The Project Team is responsible for doing the actual work and they create a Sprint Backlog (from the Product Backlog) that they believe can be completed within the 30 day sprint period.

The Scrum process is controlled through the use of the Daily Scrum meeting. These meetings are time-boxed or time limited. The Daily Scrum meeting is typically 15 minutes and only the Scrum Master and the Project Team are allowed to talk, outsiders may listen in, but are removed or silenced should they say anything.

The purpose of the Daily Scrum meeting is to answer Scrum's three questions, which are:

- What did you do yesterday?
- What will you do today?
- What obstacles are in your way?

The focus of these meetings is to quickly answer the three questions and then continue with meeting the Sprint implementation.

The Scrum process asserts that a product is never complete; after the initial construction, it is constantly under development (otherwise known as maintenance and enhancements).

2.2.2 Applicability

The majority of project management methods and techniques are prescriptive, and have a fairly fixed sequence of events with little flexibility. Scrum tries to provide a set of techniques to use that, when combined with an agile software development methodology (like XP), provides a flexible framework for project management.

Since Scrum is a project management method and does not specify any particular software development process be used, it can be adapted for use within the weapons development environment. The concern would fall upon the software development process used and the artifacts generated with that process. The best implementation of Scrum appears to be on smaller projects or even on research and development. There is a process called Scrum of Scrums that allows the Scrum project management technique to be scaled up to implement larger projects.

2.2.3 Strengths and Limitations

Scrum embraces the opposite of the waterfall approach whereby we start working on the analysis as soon as we have some requirements, as soon as we have some analysis we start working on the design, and so on. In other words, we work on small pieces at a time. This approach can be called iterative. Each iteration consists of some requirements gathering, some analysis, some design, some development and some testing culminating in an iterative release cycle (many deployments).

There are no formal project planning or roles and individual assignments. The team takes the initiative and acts within the permission it has.

Each Sprint cycle, or iteration, is supposed to be fully implementable. So, that also encompasses testing, training materials, and final documentation associated with that Sprint release.

The issue is what does that closure evidence entail? Since Scrum is a project management technique and allows the software development process used to dictate the software quality assurance evidence delivered, Scrum would have to be used in conjunction with a process that delivered appropriate SQA artifacts.

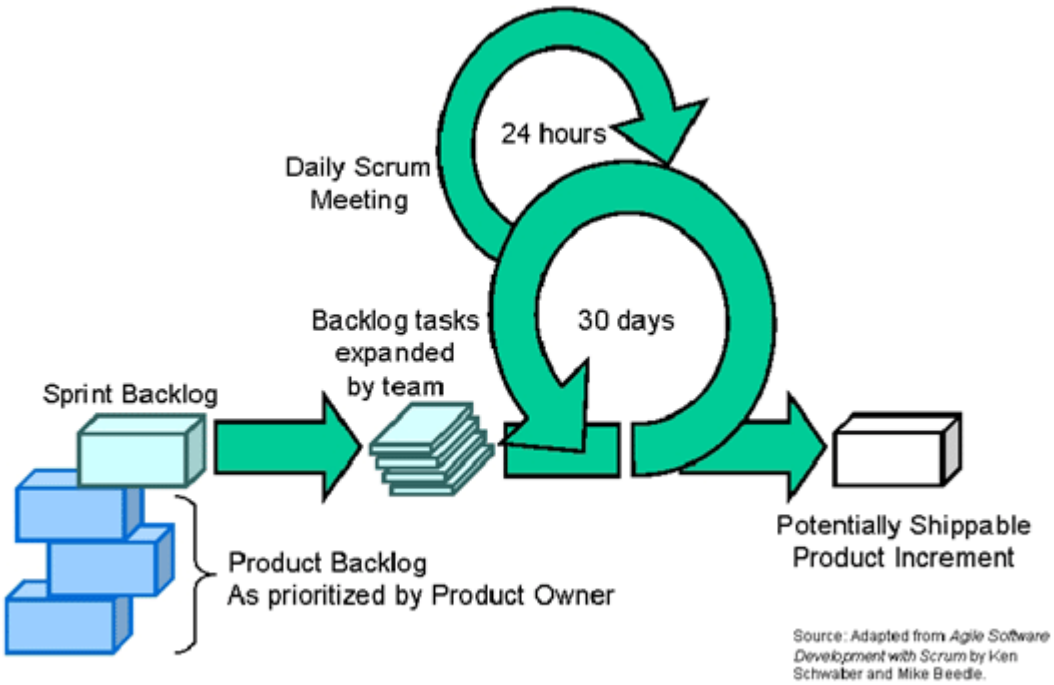


Figure 2-1. Scrum methodology

2.3 Adaptive Software Development

2.3.1 Description

The following discussion on Adaptive Software Development (ASD) refers to the methods used to develop software e.g. program management methods as opposed to Adaptive Software characterized by software that is extremely flexible using probabilistic methods or artificial intelligence, for example. ASD methods can produce an adaptive software product, but it is not necessarily a requirement. Admittedly the semantics here are confusing. See the references for some amount of clarification.

Overview – The philosophy used in ASD appears to be one of chaos theory or complex adaptive systems theory in that given complex, chaotic problems; self-directed, self-organized groups adapt to the situation and emerge to produce a product. The premise is that emergent order comprised of adaptive development, adaptive management, and adaptive concepts is favorable to that of imposed order comprised of waterfall development, command and control management, and "Newtonian scientific concepts" (term used for deterministic methods, e.g. CMM).

There are very few references found in the literature on ASD. There are one or two people that foster the approach and have documented ASD, no case studies were found, and few organizations are using it. One of the problems is in distinguishing the methodology from effective program management, or some other agile methods. James Highsmith, one of the main developers of the concept, goes to great lengths in his book *(Adaptive Software Development, A Collaborative Approach to Managing Complex Systems)* to distinguish it from Rapid Application Development and from more formal software development methods e.g. CMM or waterfall methods, but not from other agile methods so it is difficult to tell when it is ASD or something else. Highsmith uses "his definitions" of other techniques to distinguish ASD, however these definitions are not mainstream. For example, in his definition a software development plan is not changeable or dynamic to meet a changing environment or customer needs, so he uses the term "speculate" (see below). However plans are always changing, that is why it is called a plan. Similarly, requirements are always changing and developers should expect change, whether ASD methods are used or not.

The Adaptive Development lifecycle is built on a different world view or model that consists of speculate, collaborate, and learning phases (as opposed to plan, build and implement, or plan, build and revise in other methodologies).

> **Speculate** - ASD uses the term speculate to inject uncertainty and change and therefore does not try to manage projects through precise prediction and rigid control strategies. Rather "ASD strategy is more subtle – to bound, direct, nudge, or confine, but not control."[3] In complex environments, planning is paradoxical, since outcomes are unpredictable. A cohesive plan with imposed order or significant detail is viewed as non-productive. "Speculate" is used to define a mission to the best of the projects ability.
>
> **Collaborate** – Collaboration is defined as an act of shared creation or endeavor or shared discovery and is espoused as creating emergence – a property that separates the merely good from the great. Creative collaboration thrives on diversity, rich relationships,

[3] Highsmith III, J. A. Adaptive Software Development, A Collaborative Approach to Managing Complex Systems Dorsett Publishing, page 40.

unfettered information flow, and good leadership. Collaboration is manifest by interpersonal dynamic and collegiality.

Learn – Learning in ASD means, gaining mastery through experience. In an adaptive environment, learning challenges all stakeholders – developers and their customers – to examine their assumptions and to use the results of the development cycle to adapt to the next cycle.

An adaptive software development lifecycle has six basic characteristics: mission focused, component based, iterative, timeboxed, risk driven, and change tolerant.

Mission statements act as guides that encourage exploration in the beginning, but narrow over the course of a project. A mission provides boundaries rather than a fixed destination. Adaptive lifecycle focuses on results, not tasks; and the results are identified as application components. Component in this context defines a group of features (or deliverables) that are to be developed during an iterative cycle. Iterative cycles emphasize "re-doing" as much as "doing" within timeboxed bounds. The plans used for adaptive cycles are driven by analyzing the critical risks and adaptive development is also change tolerant, viewing the ability to incorporate change as a competitive advantage, not something to be summarily viewed as a "problem."

Other characteristics of ASD are living on the edge of chaos and short development cycles or time-boxes for iteration. The edge of chaos is defined as a constantly shifting battle zone between stagnation and anarchy, the one place where a complex system can be spontaneous and alive, where emergence can take place and complex products can be developed in as short a time as possible. Time-boxes are used to enforce iteration and in some respect chaos. Doing it wrong is okay as long as learning takes place.

The method recommends only limited documentation and extensive collaboration with the customer during the time-box cycles during customer focused project reviews.

2.3.2 Applicability

According to Highsmith, ASD is best suited to complex problems dealing with e-commerce, e-business or web development where time is critical and "good enough software" is developed to get to the market quickly. In the weapons development world, ASD could be useful in research activities where solutions are being explored for possible paths forward. It is not applicable for the production of product software where using the product produces substantial risk.

2.3.3 Strengths and Limitations

Strengths – ASD provides the observer a different perspective or view point on software project management. It discusses how personnel are best utilized, and emphasizes working closely with customers and developers to produce features leading to the eventual product. It discusses numerous issues like managing the challenge, development of a learning model to the evaluation of postmortem results. It is a different perspective on project management and causes the observer to evaluate projects in perhaps a different self-examining light.

Limitations – The author of Adaptive Software Development, Highsmith, goes to great lengths defining ASD primarily by assigning labels and faults to other software development methodologies. If the observer strips away all the hyperbole of chaos, dynamic, adaptive, complex systems theory, self adapting units; the rest of the document is a relatively good treatise

on program management. The author uses function points for estimating cost and schedule, but with the limited amount of documentation it is not clear how a function point is determined. Also with these methods it may be very difficult to find customers willing to fund projects when costs are unknown and not really estimated since the "speculate" phase is non-determinate.

2.4 Lean Development (LD)

2.4.1 Description

The Lean Development methodology is a proprietary approach developed by Bob Charrete[4]. Robert N. Charette PhD is currently a fellow in the Cutter Consortium and President of ITABHI, an IT consulting firm. Prior to this he worked at the Naval Underwater Systems Center at Newport, Rhode Island and at a software consulting firm Softech at the same location.[5]

The Lean Development (LD) concept evolved out of risk management principles and lean manufacturing concepts, developed by Taiichi Ohno (1912-1990) and adopted by Toyota Motors in the 1980's. Lean manufacturing maximizes adding value to the product and minimizes anything that does not directly add value to the product from the customers' view point; this includes both internal and external compliance activities. As proof of the concept, studies conducted recently indicated that Toyota US engineers reported spending 80% of their time adding value (actually doing engineering work), whereas American auto manufacturer's engineers when asked the same question reported about 20% of their time was actually spent adding value (actually doing engineering work). So there does seem to be some evidence in the manufacturing intense automobile industry that lean thinking has benefits.[6]

(1) Start Up

(2) Steady State

(3) Transition Renewal

The start up phase consists of planning, creation of business cases, and feasibility studies. Steady state is the design and build phase made up of a series of short spirals. In the transition renewal phase the product is delivered and maintained continuously. Documentation is developed and delivered in the transition renewal phase.[7]

LD contends that initiatives such as Capability Maturity Model (CMM), International Organization for Standardization (ISO), Six Sigma, Total Quality Management (TQM), and Business Process Reengineering) (BPR) have just added layer upon layer of forms, procedures, meetings, and approvals to teams desperately trying to produce something useful. Too much structure not only kills initiative and innovation, it consumes enormous chunks of time. The fundamental issue from an LD perspective is not that these initiatives are valueless, because they are not, but that they fundamentally value structure over individual knowledge and capability. LD contends that the reason these initiatives are difficult to implement is that the people feel devalued.

---

[4] Boehm, B., Turner, R., Balancing Agility and Discipline Addison-Wesley, 2004, page 171

[5] Cutter Consortium Web Page, www.cutter.com

[6] Kennedy, Michael N. Product Development for the Lean Enterprise: Why Toyota's System is Four Times More Productive and How You Can Implement It. The Oaklea Press 2003

[7] Boehm, B., Turner, R., Balancing Agility and Discipline Addison-Wesley, 2004 page 172

Lean thinking provides one way to separate compliance from delivery, which in turn enables the optimization of delivery activities. The four tenets of lean thinking are:

(1) Specify Value

(2) Identify the Value Stream

(3) Flow

(4) Pull

Value can only be defined by the ultimate customer. Value is expressed as customer need, price, time, and quality. Value always flows from the end customer, not other sources, such as internal management.

Value stream is defined as the series of activities required to bring the product to customers. Each specific activity is analyzed to see where it adds value and where it does not. Flow thinking encourages everyone, managers and staff, to abandon the traditional batch-and-queue mode of thinking for that of small-lot-flow. What ever the step, keep activities flowing from one to the next without delays for approvals or waiting for the next department. Pull thinking means waiting for customers to ask for products, rather than building a large inventory and pushing them out.[8]

Lean thinking makes one other point, while there are good reasons for some compliance activities, they need to be done in a way that does not impede progress. An example would be the rebuilding of the Santa Monica Freeway after it collapsed during the January 17th, 1994 Northridge earthquake in Los Angeles leaving millions of motorists without their daily route to work. Caltrans (California Transportation Department) engineering experts quoted 12 to 18 months to rebuild the damaged roadway. A commercial engineering firm was able to get the road open to traffic in 66 days by among other things removing compliance delays. Inspectors were embedded with the work crews inspecting as the building was being done. This saved valuable time by catching mistakes early when correction is relatively easy, and approvals were issued in real time avoiding delays and rework.[9]

The ten simple rules of Lean Programming would seem to be a software version of the same lean thinking[10]:

(1) Eliminate Waste

(2) Minimize Artifacts

(3) Satisfy All Stakeholders

(4) Deliver As Fast As Possible

(5) Decide As Late As Possible

(6) Decide As Low As Possible

(7) Deploy Comprehensive Tests

[8] Womack, James P., Jones, Daniel T., Lean Thinking: Banish Waste and Create Wealth in Your Corporation. Simon and Schuster, 1996

[9] Rubinstien, Moshe, Firstenberg, Iris, The Minding Organization, John Wiley and Sons, page 6.

[10] Poppendieck, Mary, Lean Thinking, The Theory Behind Agile Software Development, November 2002, slide 9

(8) Learn By Experimentation

(9) Measure Business Impact

(10) Optimize Across Organizations

Compliance is viewed in LD as a cost of doing business, but not something that adds value directly to the customer. Compliance activities can spiral out of control as performance shortfalls are "fixed" by additional compliance activities. Large organizations tend to put a heavy process in place to prevent low probability mistakes burdening the delivery process with controls and paperwork. LD looks at compliance as external and internal. External compliance (e.g. compliance to government regulations) should not be avoided. When the reason for complying to external regulations goes away or changes, the delivery process should change accordingly. The goal for compliance in general is to not let it interfere any more than absolutely necessary with delivery activities.

One last point of LD is that too many managers and process designers think that compliance activities – reviews, checkpoints, formal documentation, and elaborate and detailed plans – add value. One of the tenets of lean thinking is to trust people actually doing the work and push decision making down to the working level. A way to distinguish delivery from compliance activities is to ask employees "Does this activity help you deliver customer value, or is it overhead?" An example given is a design session where a series of designs is drawn out on a whiteboard by design engineers. From a product delivery viewpoint, taking a digital picture of the whiteboard and storing the photos in a project folder may be sufficient to capture the design. However from a compliance perspective (say to maintain ISO certification), the design diagrams need to be in a certain format. LD would employ support staff to put the designs into standard format, not the engineers.[11]

2.4.2 Applicability

LD is not just a software development strategy. It includes the entire chain of command. LD is a business strategy and project management approach. It is a proprietary method of Charette's ITABHI Company. Because LD is risk based, it seems to be consistent with the DOE graded approach in some ways. LD is relatively non-specific about the particular software development practices, policies, and guidelines, so it would seem compatible with a number of specific software principles and practices. Lean thinking and focusing on value to customer rather than internal or external compliance would certainly have application in a number of industries.

The LD approach would certainly seem to be a success for Toyota. At a time when all three major US automotive companies are struggling to stay in business, Toyota, using US workers to build cars in the US, is continuing to gain market share and profit. The timing and success of its Hybrid vehicle introduction would indicate that the company is very much aware of customer desires. The question is: can the techniques be ported to the DOE complex? Certainly DOE budgets are being reduced of late while at the same time explicit compliance to new standards demand more compliance for software development. Regulatory boards are adding compliance requirements and audits as they expand their span of control. The challenge may be to convince the bureaucratic hierarchy that they need to develop a lean thinking mentality to continue to have value from contractors in an era of declining budgets.

[11] Highsmith, Jim, Agile Project Management, Addison-Westley, pages 37-39

2.4.3 Strengths and Limitations

Strengths – A proven technique in the manufacturing industry. LD was developed by a noted software expert who has had experience with heavy processes as well. Because LD includes the entire chain of command instead of just a software project, it is more likely to gain sponsorship from all levels of management. LD does not dictate how to do specific software practices, so it can be combined with other agile or traditional principles and practices. There is evidence that top DOE executives are trying to eliminate bureaucratic roadblocks and therefore may be supportive of LD. LD is risk based which is compatible with the DOE graded approach. LD (or lean thinking) values people, empowers employees to add value to the product, and allows decisions to be made at lower levels.

Limitations – LD may not be embraced by a bureaucracy that has traditionally depended on heavy process for its existence. LD assumes employees are appropriate for the activities they conduct and would welcome additional responsibility. LD does not satisfy the requirements for a set of software development principles and practices; it assumes the workforce is already aware of them.

2.5 Crystal [12] [13] [14] [15]

2.5.1 Overview

Crystal was originally developed in the early 1990s by Alistair Cockburn employed by an IBM consulting group to support object-oriented development projects. Due to the majority of projects being fixed-scope and –price, the driving factor for a new methodology was *efficiency* rather than being able to handle rapidly changing requirements. Crystal is an "approach" to software development that is based on a genetic code. The instance of Crystal generated from the genetic code can vary dependent upon the needs of the software development project. Based on the criticality of the project and project size, varying Crystal methods have currently been defined ranging from Clear, for "collocated teams of 6 or fewer people," Yellow, for teams of 6-20, Orange, for 20-40, then Red, Majenta, Blue, etc. for covering greater numbers. The "heaviness" of the color associated with the methodology is used as an indicator for the level of rigor required within the methodology. In practice, it appears only three of the Crystal methods have actually been clearly constructed including Crystal Clear, Crystal Orange, and Crystal Orange Web.

The genetic code defined by Crystal consists of:

The economic-cooperative game model

(1) A set of priorities and principles for making choices

(2) Selected properties to steer toward

(3) Selected strategies and techniques

[12] Cockburn, Alistair, Crystal Clear: A Human-Powered Methodology for Small Teams, Addison-Wesley, 2004
[13] Abrahamsson, P., Salo, O., Ronkainen, J., Warsta, J., "Agile Software Development Methods - Review and Analysis", VTT Publications 478, Espoo 2002
[14] Williams, Laurie, "A Survey of Agile Development Methodologies", http://agile.csc.ncsu.edu/SEMaterials/AgileMethods.pdf, 2004
[15] Cockburn, Alistair, "Crystal Light Methods", Cutter IT Journal, 2001

(4) Project examples to copy.

The **economic-cooperative game model** is a model where each game consists of inventing and communicating. These games are usually resource limited and have two goals:

(1) Delivery of the (software) product for the current game

(2) Preparation/setup for the next game.

Each game is unique and therefore requires strategies and techniques that were slightly different from the previous game(s). The economic-cooperative game model leads people on a project to think about their work in a very specific, focused, and effective way.

The set of **priorities** that the Crystal Methodology emphasizes is:

(1) *Safety* in the project outcome

(2) *Efficiency* in development

(3) *Habitability* of the conventions

Crystal emphasizes seven "safety" properties with the first three being required.

Property 1 *(Required)*. Frequent Delivery: Frequent delivery is the single *most important property* for any project independent of size. Specifically, for software, a team should *deliver* at a minimum every four months to stay within the safety zone. Teams deploying software to the web may deliver on a weekly basis.

To understand the definition of "delivery" it is important to understand the difference between integration, iteration, and release. *Integration* of code should occur on an hourly, daily, or at least weekly basis. The term *iteration* refers to the completion of a section of work, integrating the section within the system, reporting the results to management, reflecting on the process and what should be changed or improved, and getting emotional closure on having completed the work. An iteration is normally about two weeks; but, could be as long as a quarter. Iterations are usually fixed length and the requirements are locked. The results of an iteration may not get *released*. For example, delivery may occur after every iteration, only after every few iterations, or may be based on a specific calendar date. Delivery dates must be a topic that is discussed with the entire team including the Executive Sponsor.

It is important to note that "frequent delivery" means providing the software to the users. If a product cannot be delivered to the full user base every few months, then *user viewings* between deliveries become all the more critical. With frequent delivery, critical feedback is received on a timely basis allowing all members of the team to realize actual rate of progress, early corrective action of miscommunicated requirements and errors, and last but certainly not least, implementation of the users' actual needs.

Property 2 *(Required)*. Reflective Improvement: This property allows for the team to "reflect" and "improve" and should be held on a periodic basis. A reflection workshop or iteration retrospective may be held after every iteration, on a calendar basis, or once or twice for each delivery cycle. Whatever the frequency, it is critical that the team get together and discuss working conventions *and try out new ideas* particularly for those conventions that are currently not considered successful on the project. This critical element of allowing the team to determine what is best suited for their project through continual reflection is the underlying reason for the Crystal family of development methodologies being so flexible and leaving details unstated.

Property 3 *(Required)*. Close Communication (for Crystal Clear this is extended to "Osmotic Communication"): There must be rich and efficient communication between all players of the project team. As communications moves to telephone, email, and eventually paper, the richness, speed of interaction, and emotional content diminish. Close communication can be approximated with the use of high-speed intranet, web cameras, and chat sessions. Osmotic communication is usually accomplished by seating the team in one room and infers that the communication has become such an integral part of the team that information flows into the background so that team members pick up relevant information as though by "osmosis".

Property 4. Personal Safety: Personal safety provides an environment such that any project team member can speak without fear of reprisal. This is a critical element in the enabling of communication and the discovery of project weaknesses or risks. When a person *trusts* others to not betray or damage ones' self based on the information they reveal, they will reveal more freely. This inevitably speeds the progress of the project.

Property 5. Focus: Focus involves understanding what to work on and having the time to work on it. Focus is maintained through communication for the understanding of the work to be done and an environment where people are allowed to stay on task and not be diverted to other incompatible activities. As a guideline, it is suggested that one and half projects is the maximum that a person can be on at one time and remain effective.

Property 6. Easy Access to Expert Users: Easy access to expert users provides a means for testing and deploying deliveries, rapid feedback on the quality of the current product, rapid feedback on design decisions, and up-to-date requirements. A full-time expert user on the development team is not realistic for most projects. As such, the more hours each week that expert user(s) are available, the more advantage that can be taken from their proximity and knowledge. Even one hour a week is immensely valuable and is the most critical. Therefore, when a full-time user is not an option, other alternatives for the continued access to expert user(s) include:

(1) weekly/semi-weekly user meetings with additional availability by phone

(2) developers become apprentice users

The second item of developers becoming apprentice users is a technique not widely studied. However, it appears to be a very positive method for developers to gain an understanding and appreciation of the users' job functions. The developer gains an understanding of how the final software product can change the working lives of the users.

Property 7. Technical Environment with Automated Test, Configuration Management, and Frequent Integration

Automated testing, configuration management, and frequent integration are core elements of most any agile development environment.

With respect to automated testing, successful delivery can be performed with manual testing. Therefore, automated testing may *not* be considered a critical success factor, but it certainly is an important element when reviewing methods for improved efficiency and quality of software. Regression testing can easily be performed after a revision is completed, more comprehensive testing can be efficiently performed, and a programmer can always 'retest' to be sure the code has not been broken since the last time they modified the code. This one tool will not only

improve the quality of software but can also provide a peace of mind and improved quality of life for the developers.

Configuration management is normally considered the most critical software development tool (beyond the compiler itself). Configuration management keeps software under control such that each developer knows they are working with the latest working version of the software. Configuration management allows work to be checked out asynchronously, changes can be backed out, a configuration can be wrapped up for release, and a configuration can be 'rolled back' if problems arise during deployment.

Frequent integration allows for the early detection of errors and a smaller region of code that has to be searched for the source of the mistake. In the best scenarios, there is "continuous integration-with-test".

In support of the seven properties, various strategies and techniques are deemed important by those using the Crystal methods. None of these strategies or techniques are required; however, these strategies and techniques are very helpful as a starting point in generating your own Crystal implementation. For example, reflective improvement is required by Crystal Clear; however, no specific technique (e.g., methodology shaping, reflection workshop) to implement this principle is mandated. The suggested strategies and techniques include:

Strategies:

- *Exploratory 360°:* Team reviews business value, requirements, domain model, technology plans, project plan, team makeup, and working conventions to determine if the project is both meaningful and can be delivered using the intended technology. This review aligns the team on the project's mission and approach.
- *Early Victory, Walking Skeleton, and Incremental Rearchitecture:* Using a 'walking skeleton' (a tiny piece of usable system function), users get an early view of the system and sponsors see the team delivering. The walking skeleton evolves over time to handle technology and business requirement changes. This requires the team to apply incremental development to revising the architecture as well as the system's end functionality. Overall, this establishes "early victory" in a project, supports the frequent delivery principle discussed earlier, and allows mistakes to be handled earlier in the development life cycle.
- *Information Radiators:* Information radiators typically show project status information and can be used on any project and are a tool for close or osmotic communication. However, information radiators serve the additional purpose of communicating to people external to the project. Examples of information radiators include white boards, posters, and/or monitors hung in hallways.

Techniques:

- *Methodology Shaping:* The project team interviews people from other projects and then feeds the information from the interviews into a methodology shaping workshop. The workshop generates a list of proposed rules and conventions for the development effort.
- *Reflection Workshop:* The frequency of the reflection workshop is dependent on project needs. They may be held multiple times during an iteration, at the end of each iteration, or at the end of a delivery cycle. The information from the workshop should capture

what working conventions should be retained, where problems exist, and what new or modified strategies/techniques we want to try during the next time period. This information should then be posted prominently as an information radiator.

- *Blitz Planning or Planning "Jam Session":* Using cards and poster paper identify project tasks and dependencies to develop a project map. This session is performed very early in the project and is most useful for a planning horizon of up to three-months. For longer projects, a detailed project map is produced by the business expert, lead designer, and sponsor and is then referenced by the entire project team.
- *Delphi Estimation using Expertise Rankings:* After defining the majority of use cases (or requirements), identify the factors that determine project effort (e.g., UI screens, business classes, technical framework, use cases). Senior developers then provide estimates for completion of the tasks. After three rounds of estimation, discussion is held regarding the differing factors where convergence has not been achieved. In the end, agreement on some level of effort is achieved and an understanding where there are differences. In the second phase, the type of resource (e.g., expert, intermediate, or novice developer) required for each factor is determined. This most likely will include consideration of the amount of knowledge the resource will need in the business problem domain.
- *Daily Stand-up Meetings:* A very short stand-up meeting performed daily to trade notes on status, progress, and *identification* of problems. Problem-solving should be performed outside of this meeting. Three basic questions should be answered by each participant: (1) What did I work on yesterday? (2) What do I plan to work on today? and (3) What is getting in my way?
- *Essential Interaction Design:* Technique for working user-interface design and interaction design into an agile project. Essential Interaction Design includes a workshop for initial requirements elicitation and design, deriving the user interface, usability inspections during the design, and quality assurance testing the system personalities (verification that the system personality is compatible with the user role).
- *Process Miniature:* Can be utilized to introduce a methodology (such as Crystal) to the development staff. This may require anywhere from 90 minutes to one day.
- *Side-by-Side Programming:* In side-by-side programming, two people sit close enough to easily see each others' screens, but work on their on assignments. This technique is in support of close and osmotic Communication. This technique is a variant of pairs programming where two people are working on one programming assignment at a single workstation.
- *Burn Charts:* A simple and powerful technique for the planning and statusing of a project providing visibility into a project's progress.

The Crystal methods require certain common deliverables including a release sequence, common object models, user manual, test cases, and migration code. As the level of rigor increases so does the documentation. For instance, Crystal Clear expects annotated use cases/feature descriptions for requirements documentation. On the other hand, for Crystal Orange, a requirements document is required. For the design effort, Crystal Clear products include screen drafts and design sketches. Crystal Orange demands user interface design documents and inter-team specifications. Lastly, the project management effort also has differing levels of rigor for

documentation. Crystal Clear identifies a schedule for user viewings and deliveries; whereas, Crystal Orange requires a detailed project schedule and regular status reports.

2.5.2 Applicability

Alistair Cockburn suggested two primary factors for the determination of a methodology

(1) Number of people being coordinated

(2) Damage caused from system malfunction.

To support this concept, Crystal was not developed as a single methodology; but, rather a family of methodologies that emphasizes frequent delivery, close communication, and reflective improvement. A Crystal methodology can be derived for different types and sizes of projects. Each project must use the 'genetic code' previously discussed to generate their specific Crystal methodology. The Crystal family has currently constructed methods for projects with one to forty people delivering a system with a potential loss of (C)omfort, (D)iscretionary money, or (E)ssential money. As previously mentioned, these methods include Crystal Clear, Crystal Orange, and Crystal Orange Web. Only Crystal Clear and Crystal Orange have been known to be used in practice. Crystal Clear is for very small projects with a limit of six developers that share office-space. Crystal Orange may have multiple teams with a combined maximum number of resources of forty people. It is important to note that the Crystal family currently does not support 'life-critical' systems. Neither Crystal Clear nor Crystal Orange include comprehensive design- and code-verification activities and thus are not suitable for life-critical systems.

Since Crystal defines *methodology* as a set of conventions that the team agrees to adopt and can be adapted within many software development processes, the prescribed *elements* of Crystal are viable techniques for use within the weapons development environment. However, only with substantial extension of the Crystal methods with a primary emphasis in verification activities, may the Crystal family become a viable option in and of itself for the weapons complex development.

As a stand-alone methodology, Crystal was not intended to meet the requirements of the CMMI. However, organizations that are CMMI certified should be able to introduce Crystal (or borrow from it) to increase efficiency. For example, the required properties of frequent delivery, close communication, and reflective improvement should all provide added value to a CMMI certified project. In fact, reflective improvement supports the highest CMMI capability level, '*Optimizing*'.

2.5.3 Strengths and Limitations

Strengths

- Key properties and techniques can easily be adapted within existing software development methods
- Provides guidance for deriving specific methods to be used on a project
- Supports projects with varying sizes and levels of risk
- Customer- and user-focused
- Provides techniques for project plan and task definition

- Provides techniques for requirements solicitation and definition
- Focuses developers on producing working results within a specified time period
- Risk reduction through iteration of design and build
- Provides planning, estimation, and statusing guidance
- Provides guidance for automated testing
- Provides guidance for documentation and the responsible role

Limitations

- Does not address life-critical projects
- Due to the flexibility and adaptability of Crystal, it is not as well structured or defined as some other methodologies (e.g., XP)
- Make up of the project team is critical
- Focuses on fixed-scope, fixed-price projects
- Does not address verification techniques in detail
- Restricted communication structure with co-location of team members

2.6 eXtreme Programming (XP)

2.6.1 Description

Based on the experiences of Kent Beck, Ward Cunningham, and Ron Jeffries, while at Daimler Chrysler in the early 1990s, eXtreme Programming (XP) is probably the most widely recognized agile method. One of the main goals of XP is reducing the impact of constantly changing software requirements on the software product, by focusing on delivering the simplest solutions in short iterative cycles and frequently refactoring the code to simplify and add functionality. This method is in contrast to more structured development methodologies, such as the "Waterfall" method, where requirements for the system are determined at the beginning of a project and the customer may not see the product until it is completed and delivered for acceptance testing, and any changes to the requirements in the later stages could be very costly.

XP is based on the core values of simplicity, communication, feedback, courage, and respect where:

- Simplicity means to start with the simplest solution and refactor the code to meet the customers' needs.
- Communication means to develop and implement practices that support communicating "early and often" between the customers, developers, and the users.
- Feedback means to encourage constant and continuous feedback from the customer (new requirements, reviews, testing), the system (testing) and from the project team members (issues, code reviews, bugs, changes, etc.).
- Courage means to support and encourage code refactoring, code removal, and persistence.

- Respect means to encourage the team members to respect each other's work and to strive for a high quality product.[16] [17]

eXtreme Programming advocates twelve practices that can be grouped into the following four areas: [18] [19]

Fine scale feedback

- Pair programming – the practice of two software developers combining efforts using one workstation.
- Planning game – consists of Release Planning (determine what requirements to include in a release and the release date) and Iteration Planning (tasks and team assignments).
- Whole team – the practice of bringing the whole team (developers, customers, users, testers, etc.) together for planning and tracking.
- Test driven development – the technique of writing the test case before writing the code.

Continuous process rather than batch

- Continuous Integration – the practice of frequently completely rebuilding and testing the software.
- Design Improvement (aka refactor) - the practice of frequently refactoring the software including changing the architecture and simplifying the design.
- Small Releases – consists of delivering a small segment of the total software in predetermined releases.

Shared understanding

- Simple design – the practice of taking the "simplest is best" approach to software design.
- System metaphor – the practice of naming classes/methods in a functionally appropriate manner.
- Collective code ownership – the practice that everyone on the team owns all of the code and can change any part of the code.
- Coding standards or coding conventions – consists of an agreed upon standard of rules, conventions, styles, and formats that will govern the software development.

Programmer welfare

- Sustainable pace (i.e. forty hour week) – the concept that the project team should average a 40 hour work week with overtime being the exception not the rule.

[16] Wikipedia, The Free Encyclopedia http://en.wikipedia.org/wiki/Extreme_Programming
[17] Boehm, B., Turner, R., Balancing Agility and Discipline Addison-Wesley, 2004
[18] Alexandrou, Marios, Copyright © 2002-2006 "Extreme Programming (XP) Methodology" http://www.mariosalexandrou.com/methodologies/extreme-programming.asp
[19] Jeffries, Ron, "What is Extreme Programming?" http://www.xprogramming.com/xpmag/whatisxp.htm#whole

2.6.2 Applicability

XP is best suited for use on new or prototype projects where the requirements are incomplete and expected to change frequently during the project. XP was designed for small project teams (between two to twelve team members), but it can be applied to projects with as many thirty team members. Team members must be collocated and XP requires that the customer or a customer representative be present and part of the team for input on requirements, planning, and tracking purposes. Additionally, the project domain must lend itself to test automation for an XP project to be successful.

XP is not well suited to software projects where the requirements are contractually controlled; the software is to be used in safety critical applications; the software is embedded in mass market products; the project is large and supported by many subcontractors. [20] [21]

2.6.3 Strengths and Limitations

Strengths

- Reduces the impact of constantly changing software requirements on the software product
- Focuses on delivering the simplest solutions in short iterative cycles
- Frequent refactoring of the code in order to simplify the design
- Customer is involved as part of the team
- Emphasizes teamwork, communication, and quality
- Test driven development and frequent retesting/regression testing
- Frequent reviews

Limitations

- Lacks scalability to large project teams, large software products, etc.,
- Fails to take into account project long term goals
- Lacks project documentation, such as requirements, design, etc.,
- Project team must be collocated and include the customer or customer representative
- Testing is relied upon to produce a quality product
- Practitioners frequently adopt only parts of the XP methodology and ignore the rest
- Assumes the use of object-oriented methodology
- Assumes the use of automated testing
- Informal change management

---

[20] Wells, Don, Copyright 1999 all rights reserved "When should Extreme Programming be Used?" http://www.extremeprogramming.org/when.html

[21] Emery, Patrick , "The Dangers Of Extreme Programming", A Term Paper Prepared For Swe625 Software Project Management Andswe699 Software Engineering Management May 20, 2002 http://members.cox.net/cobbler/XPDangers.htm#_Toc530042780

2.7 Dynamic Systems Development Method (DSDM)

2.7.1 Description

Dynamic Systems Development Method (DSDM), first published in 1995, is a formalized and agreed, systematic approach for Rapid Application Development (RAD) for software. It is non-proprietary and is "owned" by a consortium of around 250 organizations. The DSDM Manual, supported by two volumes of White Papers, defines the method, principles, roles, activities, and products, and offers guidance and support material. DSDM places great emphasis on customer-supplier partnership, user involvement and fitness for purpose as the driving criterion. Traditional software projects start with requirements that are often incomplete and allow time and resources to vary during development, DSDM projects are time fixed and as far as possible, so are the resources. This means that the requirements that will be satisfied are allowed to change. It is based on iterative and incremental development approaches. Activities within an increment do not follow a traditional design/code/test sequence, since all these take place in parallel. Rather, it identifies stages in which first the business needs are addressed, then the key functionality is prototyped, then other functionality and non-functional aspects are addressed. Within each stage, a number of iterations take place, using a time-box technique. Time-boxes are fixed development periods, as short as possible, which aim to implement an agreed, prioritized list of requirements. At the end of the time-box, the results are reviewed and influence the contents of the next time-box.

DSDM Life Cycle description - DSDM recommends a complete lifecycle starting with Pre-Project work all the way through to Post-Project. It recommends iterative and incremental working but with an emphasis on products or features rather than activity based (See Figure 2-2).

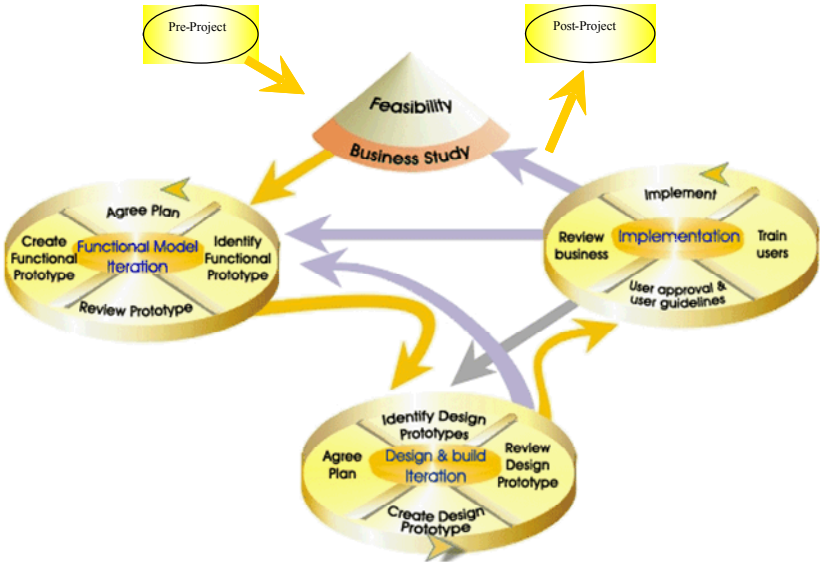


Figure 2-2. DSDM Life Cycle

DSDM Activity/Phase Descriptions:

Pre-project Activities

The manner in which projects are established will vary depending on organizational arrangements and on local working practices for the initiation of projects. Key products to be

expected from this phase include: initial definition of the business problem to be addressed; outline scope; project approval with a project manager appointed and project governance; initial plans and an appropriate strategy; initial budget and resource allocation.

Project Activities

The project consists of five phases. The first two phases are sequential and done only once. The latter three phases are iterated.

Feasibility Study

The objectives of the Feasibility Study are to establish whether a proposed development can meet the business requirements of the organization; to assess the suitability of the application to DSDM development; to outline possible technical solutions to the business problem; and to obtain estimates of timescale and costs. The level of detail should demonstrate whether a feasible solution exists or to select the most appropriate one. The feasibility report should include an outline plan and risk log.

Business Study

The objectives of the Business Study are to scope the business processes to be supported; outline the future development in terms of prototyping deliverables (defining which are incremental and which, if any, are throwaway) and prototyping controls; identify representatives of the user classes for prototyping activities; prioritize the requirements of the proposed system; reassess the risks of the project; provide a firm basis for technical development to proceed; and scope the non-functional requirements, in particular to decide the maintainability requirements. Key products include:

- Business Area Definition
- Prioritized Requirements List
- Development Plan
- System Architecture Definition
- Updated Risk Log

Functional Model Iteration

The objectives of the Functional Model Iteration phase are to demonstrate the required functionality using a functional model consisting of both working software prototypes and static models (e.g. class models and data models); and record the non-functional requirements which may not be demonstrated by the working prototype. Key products include:

- Functional Model including Functional Prototypes
- Non-functional Requirements List
- Functional Model Review Records
- Implementation Plan
- Timebox Plans
- Updated Risk Log

The requirements of any DSDM project are prioritized using MoSCoW. The o's are there for fun but they could stand for 'on time' and 'on budget'. The MoSCoW rules provide the basis on which decisions are made over the entire project, and during any timebox.

Timeboxes are fixed; therefore, the deliverables from the timebox may vary according to the amount of time left. Essential work must be done - less critical work can be omitted. So the MoSCoW rules are applied.

Must Haves fundamental to the project's success
o
Should Haves important but the project's success does not rely on these
Could Haves can easily be left out without impacting on the project
o
Won't Have this time round can be left out this time and done at a later date

A clear prioritization is developed ensuring the essential work is completed within the given timeframe.

Design and Build Iteration

The objectives of the Design and Build Iteration Phase are to refine the Functional Prototypes to meet the non-functional requirements; and engineer the application so that it demonstrably satisfies user requirements. Key product include:

- Timebox Plans
- Design Prototypes
- Design Prototyping Review Records
- Tested System
- Test Records

Implementation

The objectives of the Implementation phase are to place the Tested System in the users' working environment; train the users of the new system; determine the future development requirements; and train operators and support staff. Key products include:

- User Documentation
- Trained User Population
- Delivered System
- Increment Review Document

Post-Project Activities

The Post-Project phase contains the activities that occur once the project team has disbanded. These include support and maintenance activities and (optionally) a Post-Implementation Review to assess the system in use.

The objectives are to keep the Delivered System operational; assess whether or not the proposed benefits of the project as stated during its initial phases were achieved; enable development processes to improve; and review the Delivered System in use. Key products include:

- Post-Implementation Review Report
- Change Requests
- New releases of the Delivered System in response to Change Requests

2.7.2 Applicability

The decision as to whether or not to use DSDM is an important one to be made at the outset of a development project. Some projects are clearly ideal for DSDM. Others require careful consideration to decide whether the use of DSDM will provide business benefits.

Every organization that wants to introduce a new project approach has an existing culture and accepted working practices. Some of these will work against the effective introduction of DSDM even before the issues of whether a particular project is appropriate for DSDM are considered or whether the staff have the right skills and tool support. IT and business management should consider how to change or work around the problems as the business benefits will outweigh the costs

Within the framework there is a suitability/risk list which is a comprehensive set of criteria for helping to determine the risks that need to be addressed when applying the DSDM framework approach. Each potential project should be judged individually using the suitability/risk list. If the project provides a good match against the list, then DSDM can be considered a very appropriate development approach. However even if the project doesn't satisfy all of the criteria this does not automatically prohibit the use of DSDM, it may simply be necessary for risk management strategies to be developed to address the risks

DSDM provides a project delivery framework that can be easily tailored for each class of project tackled by an organization.

Small -DSDM is aimed at fast delivery and is very well suited to small projects. At the lower end of small there may be a need to reduce the product set and merge personnel roles.

Large - When projects are "large" some special consideration needs to be given to management and control aspects, such as increasing the formality of organizational structures, procedures, communications, etc., while keeping important DSDM concepts working, such as empowered teams.

Hybrid - Not every part of every project will be able to use full DSDM: some parts will not achieve sufficient affirmative answers in the Suitability/Risk List. These projects will then need to use DSDM and waterfall techniques in tandem to achieve the best results.

Business Change Project - DSDM is applicable in these types of projects but there are necessarily some changes to the generic framework in objectives, activities and products.

2.7.3 Strengths and Limitations

Strengths

The strength of DSDM lies in its principles as it is recognized that a framework will not hold the answers for all situations, and solutions that adhere to the Principles will most likely be robust. The Principles are:

- Active user involvement is imperative
- Teams must be empowered to make decisions
- The focus is on frequent delivery of products
- Fitness for business purpose is the essential criterion for acceptance of deliverables.
- Iterative and incremental development is necessary to converge on an accurate business solution
- All changes during development are reversible
- Requirements are baselined at a high level
- Testing is integrated throughout the lifecycle
- A collaborative and co-operative approach between all stakeholders is essential

Limitations

As with any RAD activity in an incremental framework, the final design and build may not be the most effective in terms of efficiency and ease of maintenance. Often the tight timescales and the pace do not allow for a design review and re-engineer for an optimum final solution. [22] [23]

2.8 Feature Driven Development (FDD)

2.8.1 Description

Feature Driven Development is a model-driven short-iteration process for managing the analysis, design and construction phases of a software project. Feature Driven Development was developed in 1998 by Jeff D. Luca following on the back of work by Peter Coad on Feature Lists.

As the name implies, features are an important aspect of FDD. A feature is a small, client-valued function expressed in the form: for example, "Calculate the total of a sale,", "Validate the password of a user" and "Authorize the sales transaction of a customer". A *feature* is a client-valued function that can be implemented in two weeks or less.

FDD begins with establishing an overall model shape. Then it continues with a series of two week “design by feature, build by feature” iterations.

FDD consists of the following five processes or activities:

Develop an overall model

[22] DSDM Consortium, **www.dsdm.org**
[23] DSDM Consortium, North Americas **http://na.dsdm.org/na/default.asp**

Customer representatives and development team members work together, under the guidance of an experienced component/object modeler (chief architect). Customer representatives present an initial high-level walkthrough of the scope of the system and its context. Both groups work together to produce a skeletal model, the very beginnings of that which is to follow. Then more detailed walkthroughs are conducted. Each time, the team members work in small sub-teams (with guidance from the chief architect), present sub-team results, and merge the results into a common model (again with guidance from the chief architect), adjusting model shape along the way.

Build a detailed, prioritized features list.

Using the knowledge gathered during the initial modeling, the team next constructs as comprehensive a list of features as they can in the following form: <*action*> <*result*> <*object*>

For example, *Calculate the total of a sale.*

Existing requirements documents, such as use cases or functional specs, are also used as input. Where they don't exist, the team notes features informally during the first activity. Features are clustered into sets by related function, and for large systems, these feature sets are themselves grouped into major feature sets.

Plan by Feature

The third activity is to sequence the feature sets or major feature sets (depending on the size of the system) into a high-level plan and assign them to chief programmers. Developers are also assigned to own particular classes identified in the overall object model. Weighting and sizing of each feature can also be conducted at this time. The total resources, in both time and staff can then be assessed for the construction phases. A properly developed Feature Plan can lead to incredibly accurate estimates for construction.

Design by feature

Features are bundled into manageable sets through agreement between the project manager and the chief programmer. A small set of features will be designed and built as a unit by a feature team. A feature team is typically 2 or 3 programmers.

Each feature leads to the development of a Sequence Diagram in UML. This is the design deliverable. This will be peer reviewed by the feature team against the requirements, the guidelines for architecture and development, and the design review checklist. A successful review will see the feature enter development.

Build by Feature

A chief programmer selects a small group of features to develop over the next 1–2 weeks and then executes the "Design by Feature (DBF)" and "Build by Feature (BBF)" activities. The classes are identified, and the corresponding class owners become the feature team for this iteration. This feature team works out detailed sequence diagrams for the features. Then the class owners write class and method prologs. Before moving into the BBF activity, the team conducts a design inspection. In the BBF activity, the class owners add the actual code for their classes, unit test, integrate, and hold a code inspection. Once the chief programmer is satisfied, the completed features are promoted to the main build. It's common for each chief programmer to be running 2–3 feature teams concurrently and for class owners to be members of 2–3 feature teams at any point in time.

The code for each feature is then written by the appropriate members of the feature team who own the business classes affected by the feature's functionality. Once complete, the code is delivered for peer review. When unit tests are complete and the code passes review, it will be labeled for inclusion in the system build.

2.8.2 Applicability

Feature Driven Development has been successfully used for web-based application using OO technologies. FDD is in use on internet projects across the world, and is helping those projects to deliver "frequent, tangible, working results," "on time, on budget, with agreed function".

2.8.3 Strengths and Limitations

Strengths

- Excellent reporting and planning.
- Disciplined and clear.
- Customer-focused.
- Risk reduction through iteration of design and build.
- Allows users to describe requirements in short, concise statements.
- Focuses developers on producing working results every two weeks.
- Facilitates inspections.
- Provides detailed planning and measurement guidance.
- Tracks and reports progress with surprising accuracy.
- Supports both detailed tracking within a project and higher-level summaries for higher-level clients and management, in business terms.

Limitations

- Make up of the teams is important
- High reliance on technical leads
- Does not address the tasks of gathering requirements,
- Does not deal well with User Interface design and build
- Less powerful on smaller projects than on larger ones
- Does not cover testing and deployment
- Relatively young method [24] [25]

[24] Anderson, David J., Extending FDD for UI: Implementing Feature Driven Development on Presentation Layer Projects

[25] Coad, De Luca, Le Febvre, Java Modeling in Color with UML, PTR-PH, 1999. Chapter 6

3. NWC Technical Business Practices and Weapon/Weapon-Related Software

3.1 Overview

In order to understand whether Agile Methods can be applied to weapon/weapon-related software, it is important to understand what Agile Methods are and to understand what weapon/weapon-related software product requirements are. This section provides a summary of what weapon/weapon-related software is.

Weapon Material is defined as NNSA nuclear weapons, assemblies, components, software, or parts thereof. Weapon-Related Material is defined as any material, including associated software, test and handling equipment, tooling, and fixtures, being developed and produced for or by the NNSA and intended for use in conjunction with, or in any way related to, weapon development, engineering, production, surveillance, or dismantlement.

By definition in the Nuclear Weapon Complex (NWC) Technical Business Practices (TBPs), software product is a deliverable of a product realization process. The scope of software product includes software being developed or supported for use in conjunction with weapons. For any specific system it is necessary to clearly identify what software is to be considered "product" and for each of those products which of the identification, qualification, acceptance, and delivery processes should apply. The term product is relative to the supplier and customer rather than an inherent part of the material. In one environment an operating system is a software product and in another environment an operating system is not a software product. The criteria that influence whether software is weapon/weapon-related product depend upon many factors such as:

- Application domain of the system;
- Criticality of the software functionality in support of the system mission;
- Potential for modification of the software to support system upgrades;
- Potential for reuse of the software for other system applications;
- Specialty issues such as safety, security, reliability; and
- Applicable controls such as policy and standards.

The purpose of classifying any material as weapon/weapon-related product is to provide visibility so appropriate identification, qualification, acceptance, and delivery procedures are applied. What constitutes "appropriate" will always be debated by the affected persons and agencies, so it is not possible to provide absolute answers a priori. However, it is possible to assign some "likelihood" guidance to some software. Some typical software that is more likely to be classified as product and handled in accordance with the guidelines in this document include:

- Software that is part of weapon-related material;
- Software that monitors the operational use of weapon-related material;
- Software delivered under security controls to a DOD customer;
- Software delivered to a production agency for next assembly integration as firmware in weapon-related material;

- Software delivered to a production agency for use in acceptance test equipment that tests weapon-related material;
- Software delivered as part of a Joint Test Assembly (JTA); and
- Updated releases of any software listed above.

3.2 Important Attributes for Weapon/Weapon-Related Software

Weapon/weapon-related software projects have typical programmatic characteristics that include:

(1) small number of developers/supporters (e.g., 3-5 persons) constitute the Product Realization Team (PRT).

(2) varied geographical location of PRT members, mostly closely co-located, sometimes not.

(3) matrix support for software PRT from different organizations on occasion; PRT typically includes developers, testers, quality engineer.

(4) varied types and number of users depending on application; users can be geographically close or geographically far apart; users may be next assembly or may be ultimate users; ultimate users are typically military personnel;

(5) small size of code (e.g., 2K non-commented source lines of code to 80K non-commented source lines of code, with most on the smaller side of that range).

(6) close communication with direct customers, typically system/hardware engineers; infrequent communication with ultimate customers due to geographical location of customers; customer communication typically includes some aspect of software/product capability demonstration prior to deployment of full capability.

(7) software typically provides significant functionality for which the weapon/weapon-related system is dependent, hence is considered in many cases “high-consequence”; although “high-consequence”, software is typically not used for safety-critical functions, although an argument can be made that some software is safety-related (i.e., may affect a safety-critical function in some way).

There are some exceptions to these “typical” characteristics in most projects – that is, not all characteristics hold true for every project, particularly complex software that is used for critical design or surety analyses. The following subsections provide some context for various categories of software that might result in product that is classified as weapon/weapon-related.

3.2.1 Software Directly Related to Weapons

Software developed for direct use in a weapon system, to control some direct aspect of the operational use of a weapon system, or that is an integral part of a Joint Test Assembly (JTA – used for training) is a product within the scope of this report. Any software that is a component of a weapon, either as an executable or as an embedded integrated circuit such as an Application Specific Integrated Circuit (ASIC) or in some cases as a Programmable Logic Device (PLD), is considered to be weapon software. Any such software outside the weapon would be considered to be weapon-related.

Software such as a "programmer" or a "use control coded-switch" that is part of a weapon, use control software that is used to communicate coded information to and from weapons, and software that monitors the correct operation of safety-related aspects of weapon hardware components are general examples of weapon and/or weapon-related software products.

3.2.2 Weapon-Related Test Equipment or Analysis Software

Software developed for use in testing, verifying, validating, interpreting, or analyzing weapon-related product and results is considered weapon-related software product. Software that controls the operation of acceptance test equipment for weapon material, software that is used to analyze and support decisions that affect the use of weapons, and software that validates the compatibility of weapons for use in an operational environment are considered to be weapon-related software products.

There may be circumstances that would exclude such software from being considered a "separate" software product requiring a "separate" qualification. For example, the software product for test equipment may be identified, qualified, accepted, and delivered as a part of the encompassing test equipment itself. Software product within a weapon component may be identified, qualified, accepted, and delivered as a part of the encompassing weapon component itself. An important analysis software package may be developed, used, supported, and controlled by one organization without the transfer of the product to any customer organization. In this case, the supplier and customer could negotiate the appropriate handling procedures and application specific qualification and acceptance criteria for this analysis software's use.

3.2.3 Commercial Software

Commercially available software is not usually considered to be product in the sense of this report, although there are exceptions, and all software should be controlled appropriate to its application use. Such commercial software may be used to support the application product's operational environment (e.g., operating system, run-time libraries), development or support environment (e.g., compilers, configuration control, automated drawing system software, desk top publishing, configuration control, network and security control), test environment (e.g., simulators, emulators, special purpose test software), and production environment (e.g., PROM programmers, security comparison analyzers).

However, it may be that some commercial software is bundled with software product as a deliverable product. It may also be necessary for some projects to identify and apply special "commercial buy" procedures and configuration control of support software necessary to rebuild, verify, or validate software product. On a software product material list it may be necessary to identify some development and support environment commercial software components (e.g., operating systems, run-time libraries, compilers) required to produce or control changes to the software product. Some commercial products are either directly used or bundled within application-specific "wrappers" for significant application analysis. In these cases, the commercial software will have certain requirements for qualification as a software product.

3.2.4 Programmable Logic Device

A Programmable Logic Device (PLD) is a chip whose integrated circuits have been constructed based upon a digital logic mask. The process for creating the mask information and the final PLD product and the support drawings are very similar to typical software product. In the past

the mask information and the final PLD product have not been considered to be software or firmware, but it is certainly possible to qualify PLD product as a software product. Furthermore, the distinction is becoming more fuzzy between hardware integrated circuits and software that either uses those circuits for its execution or is embedded as the circuits that defines its execution.

Typically there is no computer processor that loads or exercises the PLD products directly such as in loading and executing a program or loading data from the PLD that is used by an executable program. However, as the sophistication of integrated circuit chips and the mechanisms for creating such hardware directly from functional requirements improves, it will become even more difficult to distinguish the "software" from its hardware implementation. It is best to analyze the system functionality and how it is developed and implemented before deciding how to classify the component product as software or hardware. Thus, it may be that in some instances the PLD product may be considered to be weapon/weapon-related software product covered by this report. The cognizant PRT should carefully consider the option to develop, document, and control PLDs as software product.

3.2.5 Existing Software Product Reuse and Updates

When existing software product is reused or updated and satisfies the general definition of weapon/weapon-related software, the updated software will most likely be considered to be weapon/weapon-related software product. When software product is reused or is updated and delivered as a new product, there are always questions about which aspects of identification, qualification, acceptance, and delivery should be applied or reapplied. There are concerns whether this software product should be identified by an updated version part number or different part number. If the application or customer is different, then there is the concern whether existing handling processes can be or should be applied.

Any previous identification, qualification, acceptance, or delivery processes applied to the software will influence what should be repeated or updated. For example, if the software already has a part number and will not be changed, then the existing identification procedure may be adequate. If the software has been qualified in accordance with existing procedures or reasonably similar procedures, it may only be necessary to perform capability testing to ensure the software satisfies new application environment requirements. It may be necessary to repeat customer acceptance procedures, including the NNSA Quality Assurance Inspection Procedure (QAIP). And, new delivery procedures may be similar or very different from any previous delivery procedures depending on the new customer and the application.

3.2.6 Retroactive Application of Guidelines to Existing Software

There is always concern about what to do with existing software that has not been developed, used, and supported within a context where the weapon/weapon-related software product requirements and guidelines in TBP-306 have been applied. Examples of such concerns might include:

(1) Existing software was delivered to the customer using the shirt pocket handling technology. Are we now required to go back and apply additional qualification/delivery guidance before any further or new operational use?

(2) Existing software requires some "minor" changes, but has not been qualified in the past in accordance with the weapon/weapon-related software product requirements

and guidelines in TBP-306. Will it be necessary to apply TBP-306 for the updated version?

(3) Existing software has been delivered to production agencies through Product Specifications (hard copy documentation of system requirements, design, and test). These Product Specifications contain a small amount of software code as a hardcopy binary listing of what is to be downloaded to an integrated circuit chip. There are hundreds of such software products. Will it be necessary to change the way we do business in the future for existing product changes and new products?

The criticality of the software and expected benefits and cost of applying more rigorous methods will normally answer these questions. There must always be a balance between the expected benefits and the associated costs. From a quality perspective, the observation is that software that has been developed in accordance with reasonably good software engineering principles will not require much cost to apply the requirements and guidelines in TBP-306. Software that does not include documented requirements specifications, design information, test plans and results, and management control of versions will have difficulty satisfying TBP-306.

It is difficult to apply general rules to satisfy the above concerns. Usually such concerns are negotiated between the supplier and customer for each specific case -- or in some instances for a class of cases. As long as there is a negotiated agreement between supplier and customer, and both supplier and customer can internally handle (e.g., identify, qualify, control, use) the software, there should be fewer concerns.

3.3 TBP Requirements for Weapon/weapon-related Software

The NWC TBP System provides a full life cycle framework for weapon/weapon-related products, including software products. TBP-306 provides specific requirements and guidance for software product processes. DG10235 provides more detailed guidance and examples for interpretation of TBP-306 requirements. It is important to understand that the TBP System and TBP-306 in particular do not require specific product life cycle methodologies, tools, or methods. There are requirements for specific life cycle evidence that provides the necessary confidence that the as-built product satisfies its requirements for specific application use and has been developed and manufactured using appropriate processes. Some of this life cycle evidence is required at various times during the life cycle activities. The framework for using appropriate processes is defined by the TBP System and site- and organization-specific processes, and implemented by methodologies and tools specified by the responsible PRTs and site organizations. The general requirements for weapon/weapon-related software product include:

(1) requirement to qualify (the process of assuring that product and all associated processes are capable of meeting customer requirements) software for application use in accordance with TBPs (TBP-306); TBP-306 requirements are specifically described later in this section;

(2) requirement to cover all life cycle activities: concept, development, delivery, support, and retirement as appropriate for the application; and

(3) requirement to submit certain software product to NNSA for a potential Quality Assurance Inspection Procedure (QAIP).

The rest of this section will describe the key requirements/practices of the TBPs, specifically TBP-306, along with comparison with Agile Principles and Methods.

3.3.1 TBP-306 Key Practice Areas and Associated Requirements/Recommended Practices

The basic set of requirements as defined by "shall", "must" and "require" in TBP-306 and the associated DG10235 are illustrated in Table 3-1. There are more subtle requirements that apply, but are either referenced in other TBPs or are not stated with these words. TBP-306 is not written precisely as a "requirements" specification, but more like recommended practices written around fairly basic requirements. There are other ways to describe the "requirements" in Table 3-1 that better suite the concepts of this report. These requirements/recommended practices/guidelines are compared with the Agile Manifesto values and linked to the Agile Methods described in this report.

NWC TBP and Corporate Site processes primarily support general organizational requirements and recommended practices. Organizational Processes are covered by NWC Site Process Requirements, NWC TBP system, Site-Specific Implementation Processes, and organization/site-specific methodologies and tools. For a given software project, the Product Realization Team ensures any such organizational processes are appropriately implemented. For example, DOE/NNSA Surveys, Independent Quality Assessments, and Peer Reviews are common external and internal assessment mechanisms.

Table 3-1. Summary of TBP-306, DG10235 Shall, Must, Require Statements

Practice Area	Requirement Statement (shall/must/require)
2.1 Project Management	Each project shall have a software project plan that includes appropriate software project information.
	Software that is the result of a PRP must follow all the Technical Business Practices (TBPs) which have a major part in the implementation of the QC-1 requirements.
2.2 Concurrent Qualification	The software supplier, qualification organization, and customer must have an interface mechanism for communicating what software product is available to be qualified, the qualification criteria to be met, the applicable life cycle process phases and procedures, and the roles and responsibilities that are to be assigned.
	A conditional or acceptable status on a QER is required in order for the Department of Energy to accept software product for subsequent delivery to end-use customers or to a production agency for next assembly integration. If a software process is being evaluated, an acceptable or conditional status is required for the process to be used to produce material for acceptance.
2.3 Product Identification and Traceability	Each PQ shall be identified by a control number for callout in a using application.
	The PQ control number shall be the PQ-prefixed drawing number plus a three-digit correlation suffix (e.g.,PQ310310-00X).
	As a minimum, the drawing number, three-digit control suffix, issue and title shall be marked on the medium or medium container of the source program.
	Object programs shall be marked with the drawing number, and three-digit

	suffix of the source program from which they are derived.
	The schema must provide for an easy mapping between supplier and customer product control methods.
	The identification labels for the software product and its delivery package must satisfy constraints and requirements of both the supplier and customer.
	One or more support drawings or products listed on the product material list must have significant functional change for the two-digit version to be incremented.
	The latest issue of the source file must be maintained in machine readable form for as long as the tester exists.
	When software product that has already been released is modified, then the changes to the software product must be controlled and authorized.
2.4 Software Product Definition	It is required that software product include a qualification plan; project, quality and configuration management plans; requirements specification; design documentation; implementation source and executable code; and test plan (with test results). It is not required that all this product information be in the form of separate product items.
2.5 Design Control	A formal design process that captures, verifies, and controls software design information is required.
	Software design changes and product changes shall be incorporated using design control measures commensurate to those applied to the original design.
	Changes to software product shall be incorporated in a timely fashion and change rationale shall be documented.
	Each project shall have a software project plan that includes appropriate software project information. The plan minimally addresses schedule, resource allocation, major milestone achievements, and major task flow to achieve the specified milestones.
	Records shall be maintained of the associated product definition.
2.6 Records Management	Records shall be created and controlled to document evidence of software product production, acceptance, and delivery.
	Records shall be created and controlled throughout their life cycle to document mission functional requirements allocations to the software component and project authorizations from organizations responsible for systems, subsystems, and components.
	Records for software requirements and software component definition/ identification and any feasibility/cost analysis/trade-off studies shall be recorded.
	Records shall be created and controlled to document evidence of software product production, acceptance, and delivery.
2.7 Procurement	OTS software suppliers shall be selected on the basis of an assessment of ability to meet requirements including quality requirements.
	The required OTS software supplier quality program requirements shall be included in procurement documents and shall be evaluated to confirm supplier conformance.

	Producibility concerns of OTS software to satisfy support requirements must be addressed and appropriate supplier agreement issues (licensing, escrow, intellectual property rights, maintenance, warranty) must be identified.
	For software being procured from one NWC agency to another NWC agency, the contractual mechanisms (e.g., Interagency Contractor Order (ICO)) must be clarified and an understanding of the procurement agreement documented.
	For software procured through the ICO mechanism, the agencies involved must remain in contact to ensure requisite schedule deadlines and product/process quality are being achieved.
2.8 Delivery and Product Acceptance	The Receiving Organization must also address the accounting issues of making sure that any costs associated with the software product delivery are appropriately documented in accordance with accepted procedures.
	If there are exceptions during the QAIP, a Quality Assurance Defect Report (QADR) is issued and corrections must be made prior to completion of the acceptance.
	Exceptions to the procedures and results of the qualification activities shall be documented in the Qualification Plan.
2.9 Support	Software must be designed to be supportable (i.e., has inherent characteristics that make it easier to modify the software to incorporate requested changes).
	Software must have a defined approach (that is, a software support concept) for how changes to the software are to be managed during the support step.
	When software is to be replaced, disposed, or retired there should be some record of the action. In particular, in order to retire software existing qualification evaluation releases must be revoked so that it is clear the subject software product cannot be used in the existing operational environment.
	The development and support processes for project management (see Section 2.1), product engineering (see Sections 2.3, 2.4, 2.5), and configuration management (see Sections 2.3 and 2.4) must include mechanisms for ensuring software support concerns are addressed. Reviews should include discussion of software product supportability.
2.10 Safety	Requirements for software to support hardware that is safety-related may be identified at any stage during the PRP. When such requirements are identified, the software PRT shall incorporate appropriate safety analyses into the software definition, development, delivery, and support steps.
	The PRT shall ensure that any safety requirements, nuclear or otherwise are identified and addressed in all stages of the PRP, and that the Qualification Plan includes verification and validation activities that ensure the safety requirements are met. Participation by nuclear safety at all stages of the process is required for such software.
	The best available methods are required to ensure software safety concerns have been adequately addressed.

	The PRT shall ensure that any safety requirements, nuclear or otherwise are identified and addressed in all stages of the PRP, and that the Qualification Plan includes verification and validation activities that ensure the safety requirements are met.

3.3.1.1 Program Management

The Software Program Management Process focuses on the planning and control activities that deal with managing software product throughout its life. Software Project Planning feeds to Software Project Control via the Software Project Plan. Software Project Control monitors the project and causes Software Project Planning to re-plan as indicated by the project's statuses and indicators. The inputs from the PRP provide the foundation of the planning and re-planning. This process only ends when the software or product is retired. Until that time, Software Program Management is either in the planning or the controlling phase, but mostly iterating between the two.

Project Planning requires the organization of a Product Realization Team (PRT) and specification of a project plan that addresses: (1) Schedule, Cost, WBS, Deliverables; (2) Quality Assurance; (3) Risk Management, Requirements Management; and (4) Configuration Management. It is intended that this project plan evolve and change as necessary throughout the project. It is intended that the format and implementation of this project plan be best suited for the responsible PRT. Project Planning requirements include:

(1) PRT responsibilities: project plan and project planning evidence; risk management; quality assurance planning; qualification plan and quality plan, as appropriate; configuration management approach; project lessons learned.

(2) Project Planning Activities: Identify and document the project's purpose, scope, goals, and objectives; establish the project's software life cycle; identify the software work products/deliverables to be developed and estimate their size; estimate the resources needed; estimate the project's effort and cost; assess personnel training needs; identify the selected procedures, methods, and standards for developing and/or maintaining the software; identify and assess software risks; negotiate commitments; identify hardware milestones which require software support; identify software milestones and reviews, audits, testing, and other qualification activities; and produce a schedule that takes into account critical dependencies among groups.

Software project control addresses the aspects of monitoring software projects to verify that the software project is on schedule per the output of the planning process. It encompasses two major sub processes: tracking and controlling the project. Software project control monitors the project and causes Software Project Planning to flexibly re-plan as indicated by the project's statuses and indicators. Project Control requires the PRT to determine what results, performance, risks, corrective actions, commitments, and intergroup coordination is required. Lessons learned are part of Product Realization Report and the basis for improvement. Project Control requirements include:

(1) PRT responsibilities: flexible control and updated planning as needed; identification of results, performance, risks, corrective actions, commitments; intergroup

coordination (system and next assembly as well as customers); general requirements (cost, schedule, performance) management; identification of software support tools, compilers, and other such equipment and establish qualification pedigree and controls for such equipment; contingency actions within the project plan in case nonconformance becomes an issue.

(2) project control activities: reviewing the project performance relative to expectations; coordinating groups to accomplish tasks in an organized and concerted effort; correcting or modifying the process or the plan as results deviate from expectations.

(3) project control evidence: actual results and performance against documented and approved plans; risks to project success are identified, analyzed for appropriate risk reduction actions, and managed for acceptance; corrective actions are taken when the actual results and performance deviate significantly from the plans; corrective actions are managed to closure; commitments and changes to commitments to the customer and between affected groups are agreed to by affected groups and individuals; planning and managing of each software project is based upon the TBPs and any organizational standards; affected groups participate in identifying, tracking and resolving intergroup issues.

3.3.1.2 Concurrent Qualification

The primary areas of Concurrent Qualification that are covered by TBP-306 are qualification process activities, participants, planning, and the staging of qualification activities. It should be understood that qualification and engineering are intended to be concurrent activities, and what might be a qualification activity may be conducted by the engineering personnel. Qualification of software usually includes some combination of supplier software engineering and quality engineering defect prevention and verification activities, independent internal engineering evaluations, DOE/NNSA quality assurance surveys, and end-use customer evaluations. The top-level elements of this concurrent integration of the engineering and qualification processes are described in TBP-100 and TBP-101 (Appendix A - References). The following summarizes the requirements.

(1) qualification participants: PRT participants may include representatives from systems engineering, software quality assurance, software testing, developers, and the customers. The software PRT has the responsibility to ensure the adequacy of software definition, development, qualification, acceptance, delivery, and support.

(2) qualification activities: formation of a PRT; identification of activities and results that will provide adequate verification/validation evidence for the activities of requirements, design, implementation, test, production, delivery, qualification, QAIP; identification of software product artifacts that document the evidence.

(3) qualification documentation: activities and results are documented in a qualification plan; qualification plan may be at the software level if software is separately qualified as a product, or at a next assembly level if software is delivered as part of a next assembly qualified product; qualification plan defines quality assurance practices/methods (e.g., reviews, testing, demonstration); qualification plan identifies product attributes and measures to be addressed.

(4) qualification progress status: engineering status reports are regularly provided throughout the life cycle; progress reports are required for requirements verification, qualification plan generation and design verification, qualification plan implementation and integration verification, and product requirements validation and production.

3.3.1.3 Product Identification and Traceability

Product identification and traceability mechanisms are used to segregate, identify, and track product throughout all phases of the delivery and support steps. Records of product identification and traceability information assist in maintaining configuration control of the product. The Identification Process provides a mechanism for uniquely numbering and labeling each of the software component elements/artifacts and relating those elements to the system in which the software is to execute. The Identification Process distinguishes software product from the media on which the software product is delivered and the media on which the software is loaded for operational next assembly. The Identification Process includes an identification schema (e.g., part number, version, long name or short name, qualification and acceptance labels, and other such labels) to facilitate transfer of software product among all software suppliers and customers. The schema must provide for an easy mapping between supplier and customer product control methods. Requirements include:

(1) software product identification schema: unique artifact identification; media marking to identify the product; product qualification marking; customer packaging/delivery marking; OTS software identification; support software identification.

(2) software product media label: part number title or long name of the software product (with designation if appropriate); part number; manufacturing number including manufacturer, serial number, and production date; handling, multiple volume, customer-specific delivery or identification information; and classification information.

3.3.1.4 Software Product Definition

Software product definition includes the engineering design artifacts that incorporate interface, performance, fabrication, and maintenance requirements. These artifacts define the product characteristics required for the function, reliability, interchangeability, support, and safety of the item. The product definition includes requirements and conceptual design artifacts; detailed design, source code implementation, and unit/integration/system testing definition artifacts and associated qualification activity results; and production, acceptance, delivery definition, and final qualification of the software product. The update of product definition and re-qualification is required to incorporate modifications for corrections to defects, enhancements to functionality and performance, and changes due to environment (e.g., hardware, data) interfaces. It is required that software product include information that documents qualification; project, quality and configuration management; requirements; design; implementation source and executable code; test plan (with test results); and engineering authorization releases and changes. Requirements include:

(1) software product definition; requirements and conceptual design artifacts; implementation and testing artifacts; qualification artifacts; release and change documentation.

(2) software definition verification evidence: requirements elicitation and analysis; design definition and documentation; design configuration control; requirements traceability to design/test; requirements/design verification and validation.

(3) software engineering authorization evidence: release and change documentation.

(4) software that is part of acceptance equipment: similar product definition, although typically the software is qualified as part of the test equipment.

3.3.1.5 Design Control

A formal design process that captures, verifies, and controls software design information is required. The design process is a sequence of well-planned tasks that translate design inputs into design outputs that are then verified/validated to be technically correct and meet the customer requirements. The design control of software product is achieved through PRT management, a well-defined software design process with artifacts under project configuration management, appropriate engineering release authorization, formal release configuration management, and customer acceptance. The objectives of software design control are to:

- provide evidence of design control through complete, current, and accurate records of the design process including the capture of requirements, analysis, and engineering judgment with supporting explanations; and
- utilize design reviews, verification, and validation as the means to demonstrate attainment of design intent.

There is no requirement to use a specific design methodology, only to accomplish the required evidence of design. Typically a methodology to achieve that design development and control is used. Requirements include:

(1) PRT organization: defines and implements a design quality management approach.

(2) design control process: controls project planning; requirements management; design engineering.

(3) verification and validation evidence control: reviews (e.g., peer, system, management, customer), product testing, acceptance testing.

(4) configuration management system: NWC drawing system and associated for control of all release versions of software product artifacts.

(5) engineering authorization system: NWC engineering authorization system for control of all releases and updates to versions of software product artifacts.

3.3.1.6 Records Management

Records Management provides methods for identifying, collecting, filing, maintaining, retrieving, distributing, and retiring records that furnish objective quality evidence. Records document evidence of software product production, acceptance, and delivery. Acceptance

information is captured in a variety of records such as internal qualification verification reports, qualification plan, qualification evaluation release, and external DOE/NNSA Quality Assurance Inspection Procedure report. Customer evaluations provide a level of acceptance for software product and associated operational manuals that will be used by military customers. Requirements include:

(1) software project planning records: schedule, cost, plans, resources, reviews, lessons learned.

(2) software product engineering records: requirements, design, implementation, test, build.

(3) software product qualification records: qualification plan/results, production, acceptance, and delivery.

(4) software record storage: software product definition records in standard storage repositories such as the NWC Drawing System; software project records in controlled storage repository.

3.3.1.7 Procurement of Off-The-Shelf and Delivery of Interagency Software

Off-The-Shelf (OTS) software that will be procured and used as part of the operational system or the support environment for an operational system requires appropriate risk management and evaluation activity. OTS software may be procured from a commercial vendor, industry partner, or a government agency. A Supplier Survey or more formal assessment of a vendor's Software Quality Program may be appropriate. The procurement of software product through an ICO is primarily achieved through PRP development followed by software product delivery. Negotiations during pre-production and production activities establish the product to be delivered, level of product qualification and acceptance, delivery inspection responsibilities, and product delivery authorization. Requirements include:

(1) software verification/validation of application requirements: OTS procurement of software to be used in a weapon/weapon-related application requires verification/validation evidence that the software application requirements are met.

(2) commercial software vendor survey: OTS procurement of software to be used in a weapon/weapon-related application may require a Supplier Assessment/Survey as well as satisfaction of Quality Program Requirements (e.g., elements such as: organization, documentation, and records implementation; software process improvement program; software engineering processes; software quality engineering processes; procurement program/vendor agreements; support capability for updated releases and response to problems; and management and software personnel capabilities).

(3) interagency software delivery: an Interagency Contractor Order (ICO) Software Delivery requires an Interagency Contractor Order (ICO) for qualified software product delivery; adherence to a shipping and receiving process as documented in the ICO; software product delivery shipper and product labeling in accordance with the DOE/NNSA QAIP process.

(4) shipping organization procedures: transfer the product from an internal control mechanism; verification the product has been adequately identified, qualified,

accepted, and that delivery authorization paperwork is in place; packaging the product; verification the packaging is appropriately labeled; and transfer the product to the transportation mechanism.

(5) receiving organization procedures: inspect the package for shipping damage; check that the package has the proper transfer paperwork (e.g., ICO); ensure that all security handling issues have been or are being addressed; verify that the package has been appropriately stamped; inspect the software product within the package for shipping damage; verify that the software product has been properly stamped and labeled with the correct part number, serial numbers, and identification names in accordance with the transfer paperwork (e.g., ICO); verify that the software product information has been released by the originating organization; and ensure paperwork is completed and the software product is securely transferred to the appropriate internal control storage location for subsequent use.

3.3.1.8 Product Acceptance

Product Acceptance defines the software product acceptance activities performed to assure compliance to specified requirements. Acceptance actions are applied to software product that is developed as part of a Product Realization Process as well as software product that is procured from vendors Off-the-Shelf (OTS). Acceptance activities can be classified as internal to the PRP, internal to a site, and external between a site and its various customers. All acceptance activities provide the team of suppliers and customers with assurance that the integrated software product conforms to its requirements. Product acceptance requirements include:

(1) PRP qualification acceptance activities: requirements demonstration/evidence review; requirements documentation; requirements testing results; qualification activities requirements review; product release and completion review.

(2) Released product: product released to NWC drawing system (IMS) for NWC configuration management; formal identification and release process; formal change control management process; records management for formal audit.

(3) Non-Conformances/corrective actions procedures: non-conformances/corrective actions may be discovered during: PRP Qualification Acceptance Activities; Internal Site Production Acceptance Activities; External Site Acceptance Activities (DOE/NNSA, DoD Ultimate Customer).

(4) Internal site production acceptance: final independent review of the software product definition and qualification evidence; verification of software product definition released to NWC configuration management system; determination that software product quality evidence is acceptable for operational use of the software within its application domain; official submittal of software product to DOE/NNSA for QAIP.

(5) External site acceptance: final assertion of the software product quality and that the software product is acceptable for operational use through DOE/NNSA QAIP and/or DoD Ultimate Customer Review.

(6) Product shipping/receiving: shipper, delivery agreements (e.g., ICO specification); DOE/NNSA inspection; DoD Ultimate Customer shipping/receiving inspection.

3.3.1.9 Product Support

Software supportability is the key to a cost-effective infusion of new technology into weapon components with minimal change to the weapons. Such software must be designed to be supportable (i.e., has inherent characteristics that make it easier to modify the software to incorporate requested changes). Such software must have a defined approach (that is, a software support concept) for how changes to the software and updated software delivery and installation are to be managed during the support activity. Specific project activities result in software that has been designed to be supportable and an approach/concept for how software is to be supported during its operational use. Software support requirements include:

(1) software project planning and oversight: addresses software supportability as an integral part of development process; addresses how software product support evolves from/during software development practices.

(2) software quality: addresses sustainment of QA activities and software documentation during the support activity.

(3) software engineering environment: addresses sustainment of software engineering support systems during the support activity; addresses sustainment of software engineering methods/techniques during the support activity.

(4) software support concept: defines in-service support activities that include change management; change scenarios; roles and responsibilities for change; design of the software for change; failure identification and corrective action; validation and acceptance of changes; configuration management approach during support; packaging, storage, and delivery of changed product; retirement of the software product from operational use.

3.3.1.10 Safety

Software that is part of a system implementation that is critical to nuclear safety must be analyzed as a normal component of the system safety program. Software should be a part of the system's safety program, which should include information equivalent to a safety plan and safety case. The plan provides a view of the safety-specific tasks and processes that are to be conducted as an integrated part of the normal software and system life cycle phases. The case documents that required processes and tasks have been conducted in accordance to required policy and standards and that the results of the safety-specific tasks provide adequate assurance that the safety requirements have been met. Any specific risk issues or possible safety related problems are documented in the case. Software safety requirements include:

(1) roles and responsibilities: PRT applies system safety terminology appropriate to safety-specific activities that might be applied to software; design and qualification of such software, including revisions to the software, is subject to independent review and approval by nuclear safety.

(2) identification of safety-related software: PRT identifies system allocation of safety-specific functions to software; software that controls or monitors hardware that is safety-related is so designated in design drawings and specifications; software that provides critical analysis that affects the design or acceptance of safety-related product may be considered safety-related.

(3) safety software framework: task activities are defined in a "Plan" that may be part of the qualification plan; task results are defined in a "Case" that may be part of the qualification plan results.

(4) safety management: organization, responsibilities, resources, staff qualification and training; use of a software life cycle model; configuration management; metrics; tool support and approval; documentation requirements for the plan and case; certification method; Off-the-Shelf software considerations.

(5) safety analysis tasks:

1. Identify system safety requirements and their allocation to possible software components.
2. Conduct system preliminary hazard analysis.
3. Review identified hazards and identify, categorize, and prioritize the top-level software safety hazards.
4. Apply appropriate design architecture methods to reduce and isolate the risk of the hazards within the combined hardware/software design solution. Design for ease of supporting the safety-related software.
5. Analyze the safety-related software components to determine if the design or code implementations have possible fault paths that could activate any of the identified hazards.
6. Analyze fault trees for the safety-related software components and participate in the software test design to ensure adequate test cases for the safety-related software across the defined fault trees.
7. Analyze the safety-related software test results to ensure all software safety requirements have been met.
8. Analyze software production and delivery procedures to ensure no safety hazards are introduced.
9. Analyze any changes to safety-related software for possible safety impact.
10. Document software safety analysis and test results in a software safety case to support qualification, acceptance, and certification activities.

(6) safety considerations during software support: change analysis, safety re-analysis implications, design for safety supportability (e.g., use of safety kernels), maintenance of the safety plan and case.

3.3.2 Agile Method Values and Principles Applicability for Weapon/Weapon-Related Software

This section provides a brief analysis of how the requirements of weapon/weapon-related software as summarized in Section 3.3.1 compare with the Agile Methods values, principles, and the method instances as summarized in Section 2. An illustration of the relationship is shown in Figure 3-1.

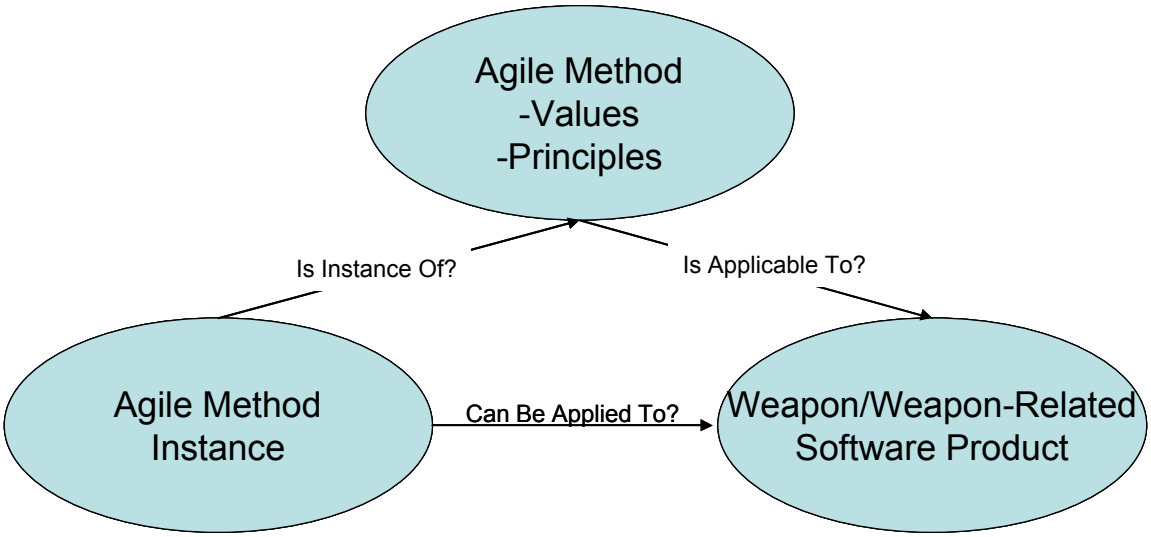


Figure 3-1. Agile Method and Weapon/Weapon-Related Software Relationships

The key questions to be answered as suggested by the Figure 3-1 are:

(1) Which Agile Method values and principles are applicable for Weapon/Weapon-Related software products?

(2) Which Agile Method instances from Section 2 can be applied to Weapon/Weapon-Related software products?

(3) Which Agile Methods from Section 2 are actually "instances" of the Agile Method values and principles?

The first two questions will be addressed to some extent in this section while the third question is addressed to some extent in Section 2.

3.3.2.1 Agile Method Values, Principles Applicable to Weapon/Weapon-Related Software

3.3.2.1.1 Agile Manifesto Values

Note that the Agile Manifesto "values" provide a "relative" hierarchy, not an absolute one. That is "valued over" does not mean "does not value", but simply provides a relative ranking of the preferences. Thus, the values do not say that processes and tools are not valuable, or that comprehensive (or perhaps a more acceptable term – requisite) documentation is not valuable. Of particular note is the fourth value – capability to respond to change rather than "blindly" following a plan (meaning a rigid, non-changeable plan). The basic premise of developing software in the first place is to make sure the software has the capability to be changed – which also means the software development/operation/support processes should also have the capability to accept change efficiently throughout the life cycle. Software and its associated processes and tools and documentation and contracts and plans must be able to adjust to changes. That is a very valuable "value". What one must be very careful is the interpretation that Agile software development means the predicates on the right side of the value equation have no value.

The Agile Manifesto values and principles are more philosophical statements than implementation practices. There are also certain assumptions built into the values and principles that may not be true (or at least possible) for some applications. It is in the mechanism for implementing practices that adhere to the Manifesto (i.e., Agile Methods) where the difficulty has always been. In particular, these Values and Principles have been known for some time and implemented in varying degrees through past methods and techniques. In fact, the Agile Methods described in this report are perfect examples of the range of interpretation that such implementations can represent. And, such "Methods" do indeed incorporate many of the existing methods and techniques as supporting practices.

V1: Individuals and interactions are valued over processes and tools. Chief Programmer Teams in the early 1970s recognized the value of the Value 1 above – small interactive teams, co-located for ease of communication and rapid evolution of ideas, prototypes, and the eventual final product. Of course, this did not necessarily mean that processes could not be used or that "tools" were not valuable. It is certainly difficult to imagine a software development effort without a compiler, or in more recent times a full development suite of tools to assist the rapid turnaround of prototypes and conceptual solutions.

V2: Working software is valued over comprehensive documentation. The Value 2 describes "working software" as being valued over "comprehensive documentation". Clearly if the deliverable software does not work, it won't matter too much what documentation has been developed – unless of course there is the opportunity to make changes so the software does actually work. Working software is not a well-defined term, but reflects more of a philosophy that what is important is to deliver software with the capabilities (working) that provides what the customer values. The type, quantity, and style of documentation produced depends on the customer, what evolution in the software product is expected/possible, and the criticality of just what the working software capability is intended to be. It is difficult to generalize such a value and declare specifics of what documentation is valuable or not, but it is important for the software team to understand the dynamics of what documentation is valuable (for them throughout the life cycle – not just development, and to the customer as well) to produce.

V3: Customer collaboration is valued over contract negotiation. Although Customer Collaboration and Contract Negotiation appear to be on polar sides of Value 3, it is unclear why there can't be Customer Collaboration and Contract Negotiation. There has always been both on most contracts. So, the emphasis is on having the proper balance and that perhaps it is better to have more customer collaboration to make sure that any contract negotiation is not only reasonable, but rather accurate as to what is desired. Close Customer Collaboration certainly should facilitate any Contract Negotiation – assuming the Customer and the Contracts personnel have some relationship.

V4: Responding to change is valued over following a plan. As for Value 4, one key attribute of a **highly** mature organization is the capability to flexibly respond to change, e.g., change their processes during any project in order to optimally achieve project performance, cost, and schedule results. Relative to the CMMI, this is defined as a Level 5 organization, but clearly does not preclude following a "plan". The implication is that once a "plan" is defined there is no way to change it. Most existing "common" software development methods clearly allow for iterative change. So, the question again is not one of allowing ad hoc change without any planning, but of establishing a planning approach appropriate for the specific application project – where the likelihood of change is incorporated.

The general attributes of weapon/weapon-related software products described in Section 3.2 and 3.3.1 provide some boundaries for what the balance of these four Values must be. For each specific application, the boundaries will tend to vary – some more strongly toward the left side of the value statements, some more strongly toward the right side of the value statements. This balance will dictate more precisely which specific Agile Method (e.g., SCRUM, ASD, LD, Crystal, XP, DSDM, TSP, FDD), if any, might have enough common features to justify their use during the life cycle of a specific weapon/weapon-related software product.

Bottom Line: In summary, there is nothing inherent in the Agile Method value statements that can not be applicable to weapon/weapon-related software, but in reality, both sides of the value statement are required parts of a weapon/weapon-related software project. The balance of what "valued over" means is what determines how applicable the values are.

3.3.2.1.2 Agile Manifesto Principles

Each of the principles will be briefly discussed in relationship to the weapon/weapon-related software requirements summarized in Section 3.3.1.

P1: Customer is highest priority – early and continuous delivery of valuable software: this principle is really two parts. First, there is no question that for weapon/weapon-related software that the customer is highest priority. Customer requirements are critical and adherence to satisfying those requirements is of highest priority for success. The second part about early and continuous delivery of valuable software makes the assumption that this is the customer's highest priority. In weapons applications it is not possible, nor desirable by the customer, to have early and continuous delivery of software (valuable or not). However, there are frequent customer interactions where early versions, user interfaces, and other aspects are discussed and demonstrated. Also, there are instances where development versions of the weapon/weapon-related software are usefully used by the next assembly systems engineers for design analysis – but not for actual product delivery. Also, typical weapon/weapon-related software delivery occurs only once as a qualified product. Any planned updates, changes, improvements to the software that requires another delivery only occurs for certain applications.

Bottom Line: part of this principle is applicable to weapon/weapon-related software projects, but the premise of early and continuous delivery of valuable software is not generally applicable. Clearly there are emergency situations where the risk of rapid delivery must be balanced with assurance of a valuable product.

P2: Welcome changing requirements, even late in development.

This principle represents more of an attitude than a principle. The principle should be that requirements typically do change, even late in development, and the project should be organized in such a way as to be able to rapidly analyze such changes, determine project risk, and make decisions as to whether such changes can be implemented. If the premise of the principle is that a software project should be able to always adapt to any change in requirements, then this would not be applicable to weapon/weapon-related software projects.

Bottom Line: this principle is applicable to weapon/weapon-related software projects in that such projects should be able to adapt to changing requirements that are essential to the customer. This may require more labor resources and a longer schedule in some instances – the goal being to have a high enough project maturity to minimize the cost/schedule impact of such changes.

P3: Deliver working software frequently.

This principle is a version of part of the P1 principle and was covered in that principle's discussion.

Bottom Line: this principle is not generally applicable to weapon/weapon-related projects.

P4: Business people and developers must work together daily throughout the project.

This principle is interpreted to mean stakeholders, customers, and others who must interface with the software product must work together. The possible problem for weapon/weapon-related software projects is the frequency of interactions. In some weapon/weapon-related software projects the customer is co-located and communication is daily. In other cases, the customer may be several continents away – and daily interaction is simply not practical. However, as a general rule this principle is a good one.

Bottom Line: this principle is applicable to weapon/weapon-related software projects, but may not be practical in all cases.

P5: Build projects around motivated individuals, good environment and support, trust.

Bottom Line: this principle is applicable to weapon/weapon-related software projects without much discussion. However, one must be careful not to interpret this to mean reward the "hero".

P6: Face-to-face conversation is the best communication method.

This principle sounds good on the surface, but one must not make it appear like this is the only communication method. The implication of this principle is that perhaps written documentation is not a good communication method. Under certain circumstances and project timing, it is clear that close, face-to-face, daily communication is required. Under other circumstances, it is also clear that if it isn't written down, it doesn't exist. It is required that documented evidence of weapon/weapon-related software is produced for communication with NNSA acceptance personnel that the software evidence exists. Face-to-face conversation is not the best communication method in this instance. Other examples exist as well.

Bottom Line: this principle is applicable to weapon/weapon-related software projects, although there is some balance required as to when face-to-face is "best" and when other communication methods are "best".

P7: Working software is the primary measure of progress.

This principle is a good one, as long as the term "working" is clearly understood. Since the term "working" is relative, both in capability and in relation to who does the evaluation, the principle is open to some interpretation. However, it is clear that if the software does not accomplish what the customer desires, other evidence will not be convincing. Of course, the "other evidence" may allow for software that does not satisfy some customer-desired capability, to be more easily updated to satisfy that capability.

Bottom Line: this principle is applicable to weapon/weapon-related software projects, but again demands a balance between the types of evidence that is developed to support the claim that the software is working, as well as to support changes to software that is working but to which capability needs to be added. Of course, this concern is not unique to Agile Methods or weapon/weapon-related software.

P8: Agile processes promote sustainable development. The sponsors, developers, users should be able to maintain a constant pace indefinitely.

This principle is based on the concept of the P3 principle of delivering working software frequently, and appears to reflect a general organizational behavior where reasonable demands are made for continual improvements over an indefinite time period. However, this principle does not define what "sustainable" means in terms of an on-going support concept (different from development), but only in terms of a continual, and regular (constant pace) development effort. In fact, the "sustainable" aspects that are required relate to building the software to be supportable and sustainable, during development as well as support. It is not clear that this agile principle addresses "sustainable" in terms of a support concept, but only in terms of continual development.

Weapon/weapon-related projects are not funded and scheduled with a constant pace or indefinite development in mind. There tend to be periods of time where this principle (i.e., the constant pace part) is true, but since weapon/weapon-related projects are focused on stockpile products (for the most part), the "indefinitely" part of the principle does not apply and the "constant pace" tends to change over time with the demands of the project and customer-directed modifications. However, weapon/weapon-related projects generally do not adequately address the sustainable/support concepts that might eventually result in a capability to better achieve the intent of this Agile principle.

Bottom Line: this principle is interpreted to relate to the agile principle of frequent delivery of working software as a means to achieve the sustainability and constant pace of development. In that respect, this principle is deemed not applicable or in fact true for weapon/weapon-related software projects. However, as the principle applies to sustaining a reasonable work load throughout the development effort (although the pace and effort may still vary significantly), that part is applicable. The "indefinite" aspect of this principle is generally not applicable and needs clarification as to the similarities and differences between development and support activities. A better sustainment/support concept for all weapon/weapon-related projects would make it more reasonable to achieve this Agile principle over an extended period of time and across multiple related projects.

P9: Continuous attention to technical excellence and good design enhances agility.

Bottom Line: this principle is partially applicable to weapon/weapon-related software projects, in that the capability to rapidly adapt is directly related to technical excellence and good design.

P10: Simplicity--the art of maximizing the amount of work not done--is essential.

This principle values the balance between what is termed "scrap" – that is, work that is either not valuable toward producing the product or re-work that has to be done because the product has to be re-done. However, simplicity may or may not relate to this maximization. Simplicity – or the minimization of the count of objects – is a very important characteristic of a project.

Bottom Line: this principle is applicable to weapon/weapon-related software projects, although simplicity is also a relative term and must be applied relative to each specific application.

P11: The best architectures, requirements, and designs emerge from self-organizing teams.

This principle is really more philosophy than the technical evidence and research can support. It is not clear just what the scope of a "self-organizing" team is. Perhaps this means people who

have a common focus and concept get together because they are motivated to solve a problem or class of problems. Second, it is not clear why such teams would be interested in architectures, requirements, or designs in particular. What is conceded is that motivated and skilled teams are more likely to produce quality products – but even those teams can be defeated by outside influences. Also, it is important to allow the talents and capabilities of team members to be integrated into the team roles, since that will be more likely to produce the best contributions from each of the team members. In weapon/weapon-related software projects, the teams simply are not "self-organizing", but there is the capability for teams that are organized to morph their roles and responsibilities to fit the project needs.

Bottom Line: this principle is applicable to weapon/weapon-related software projects only in the sense that project teams (PRTs) have the flexibility to adapt the roles and responsibilities, including adding new members or releasing members whose capabilities do not fit the project needs.

P12: At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

This is an excellent principle for any project.

Bottom Line: this principle is applicable to weapon/weapon-related software projects.

3.3.2.1.3 Agile Method Instances

This section provides a brief summary based on the information presented in Section 2 and the associated references as to whether the identified agile method instances have some or any applicability for use in weapon/weapon-related software projects.

In general, there are several conclusions that can be made based on the review information presented in Section 2 and the weapon/weapon-related software attributes and requirements presented in Section 3.

A summary of the listed Agile Methods and their more applicable and less applicable practices/features is illustrated in Table 3-2.

Table 3-2. Reviewed Agile Methods and Their Applicability to Weapon/Weapon-Related Projects

	Applicability of Agile Method Practices to Weapon/Weapon-Related Software Projects		
Agile Method	**Applicable Practices**	**Not Applicable Practices**	**Overall Assessment of Applicability**
Scrum	• daily Scrum • iterative concept • allows integration with software development method	• informal project planning • inability to integrate with software development methods that are acceptable • lack of "critical" systems methods	Limited: • some project team interaction concepts are applicable • does not directly address software development / support methods
Adaptive Software Development	• extensive collaboration with customers • basic phases (speculate, collaborate, learn)	• short development cycles, except as used as a prototype • limited documentation	Limited: • very little evidence of use • targeted domain is e-

	Applicability of Agile Method Practices to Weapon/Weapon-Related Software Projects		
Agile Method	**Applicable Practices**	**Not Applicable Practices**	**Overall Assessment of Applicability**
(ASD)	• six characteristics of adaptive development cycles • focus on program management	• "good enough software" philosophy with "learning" to correct deficiencies • production software, lack of "critical" systems methods	business • might be useful for research software and/or concept development, but not for production software
Lean Development (LD)	• systems management chain of command concepts • ten simple rules are ok • allows integration with software development method • value stream approach to customer satisfaction • external compliance should not be avoided	• more of a management philosophy than a software development approach • employing support staff to translate draft information into documentation artifacts • focus on "end" customer may limit meeting multiple stakeholder requirements • lack of "critical" systems methods	Limited: • good evidence of manufacturing project management success • does not directly address software development / support • proprietary aspects may limit use
Crystal	• genetic code • three priorities • seven properties (except for property 1: frequent delivery • three strategies • nine techniques • scalability	• required property 1: frequent delivery • lack of comprehensive design and code verification activities • lack of "critical" systems methods	Medium to High: • some verification activities would need enhancement • critical systems would need to add specialized practices • Crystal Orange for production • Crystal Clear for research
eXtreme Programming (XP)	• five core values, except for some interpretations concerning refactoring • fine scale feedback practices • shared understanding • programmer welfare (as stated)	• continuous process (frequent delivery, continuous refactoring, small releases) • lacks scalability to large projects • co-location of project team and customer • informal change management • test-centric verification • lack of "critical" systems methods	Limited: • the almost total focus on frequent delivery, and the activities around frequent delivery (e.g., co-location of customer and team) make XP limited in its use • might be used in research and concept efforts, but the co-location aspect is still not viable
Dynamic Systems Development Method	• life cycle phases • MoSCoW Prioritization • scalability (large, hybrid, small) • nine principles (most)	• principle: active user involvement – needs flexibility to allow for non co-location • principle: frequent delivery	High: • provides detailed practices that can be generally tailored for use in

	Applicability of Agile Method Practices to Weapon/Weapon-Related Software Projects		
Agile Method	**Applicable Practices**	**Not Applicable Practices**	**Overall Assessment of Applicability**
(DSDM)	• roles and responsibilities • suitability/risk criteria for application determination • large user community; company consortium; • successful case studies and supporting research evidence	– needs flexibility to mean prototyping with user involvement, not "release" of product • artifacts: detailed documentation templates may need to be tailored for weapon/weapon-related software applications • lack of "critical" systems methods	weapon/weapon-related software projects • flexible application of principles makes this method applicable
Feature Driven Development (FDD)	• design and build processes • roles and responsibilities • eight basic practices (note – these practices need to be added to the FSD section)	• lacks strong requirements elicitation or full life cycle approach • lack of "critical" systems methods (although claims to be able to be used)	Limited: • could be integrated within a more comprehensive life cycle approach • probably more useful in prototyping and research applications

4. Executive Summary

4.1 Barriers to using Agile Methods for Weapon/weapon-related Software

This section addresses barriers to using agile methods for weapon and weapon-related software. It will be noted that some barriers may be unique to the Nuclear Weapons Complex (NWC), while other barriers are applicable for any high tech or highly regulated business such as pharmaceutical industries, nuclear power plants, and aircraft manufacturers.

A number of barriers have been identified in the previous sections for the use of agile methods in weapon/weapon-related software. These barriers include large and complex software programs such as those for ASC, software teams and customers that are not collocated, and the need to adhere to software configuration management and change control for complete traceability. Most barriers do not preclude the use of discreet pieces of agile methods for rapid prototyping followed by a more structured, requirements traceability approach. Agile methods may be applicable, for instance, at the beginning of a development project when War Reserve controls are not strictly needed or for research software such as many of the ASC codes. However, there is always risk associated with any project that goes from an early concept development to full scale WR development or, as with ASC codes, is used for significant decision-making as part of the full scale WR development. In these high-consequence cases it is absolutely necessary to provide adequate evidence for the assurance of hardware and software products. Otherwise, the product, including software, may not be "sellable" to the customer, as determined by NNSA, unless formal documentation **is** part of the product. There is a saying around the NWC: If you don't have formal, written documentation to support development and production processes and product acceptance, then it didn't happen.

Sections 3.3.2.1.1 and 3.3.2.1.2 explore the four values and twelve principles which comprise the backbone of agile methods and are deemed the Agile Manifesto. The two values – working software is valued over comprehensive documentation, and responding to change is valued over following a plan are briefly discussed. Comprehensive documentation many times is a vital part of the product for weapon/weapon-related product and could represent a barrier to use of agile methods. This is not unique to just the NWC, but applies to most companies with high consequence applications. For example, pharmaceutical companies rely on statistical software to provide proof to the FDA that the drug up for approval can weather the scrutiny and test of time. Any mistake may be fatal for the public that consumes these products. Assuring software reliability typically requires significant documented evidence of due-diligence. It should also be understood that a plan is a living set of documented information that is meant to be changed – perhaps frequently. Agile methods appear to assume that a plan is a static document. Plan the work and work the plan is a solid philosophy by which to run a weapons project, or in fact, any project. The life of a weapons program is extremely long in terms of agile methods. When changes are made to a weapon, software may be affected. Longevity of the weapon and related software and the capability to make changes throughout its long life, as needed, are dependent on the availability of comprehensive documentation.

Principle one addresses two concepts: the customer as a highest priority and early and frequent delivery of valuable software. The first half of principle one is correct – the customer is the highest priority. However, the second part does not necessarily follow from the first part. In particular, early and continuous delivery of any (even valuable) software can not be supported by the typical NWC customer, and therefore is not practiced by the sites within the NWC. This

barrier may be unique to the NWC, but most customers who have product to produce can not assimilate frequent changes to their software. This is further discussed in section 3.3.2.1.2. The discussion of principle one is also related to principle three (Deliver working software frequently) and principle seven (Working software is the primary measure of progress). Principle three is in fact a subset of principle one. The delivery of working software is clearly a requirement for weapon/weapon-related software. However, if the definition of "software" as used by agile methods simply means code, this would be a barrier for using agile methods. If "working software" includes comprehensive documentation then such a delivery would be a good measure of progress. However, the concept of "progress" for agile methods is tied to the concept of frequent delivery rather than qualified product delivery as required by weapon/weapon-related software. In examining articles regarding agile methods, the delivery of software typically does mean only the delivery of code that works.

Principle two addresses the capability to incorporate changing requirements even late in development. This may present a challenge to projects within the NWC. Receiving changes even late in the project should be handled and not met with trepidation, but most organizations are not mature enough to do that. It is also not clear that organizations using agile methods (except perhaps in low consequence, small projects) would be able to handle many changes "late in development" – without cost and schedule implications. Either the software development will be short changed in performance or the schedule will be lengthened with increasing costs.

Principle four addresses the concept that business people and developers must work together daily throughout the project. In theory, this should not be a barrier. From a pragmatic point of project management, there may be issues. If the word daily were not a part of this principle, then the principle would not be a problem. Weapon/weapon-related projects frequently involve internal and external customers who continually work together, but can not communicate daily. Principle five addresses sound concepts of building projects around motivated individuals, good environment and support, and trust. Clearly these are excellent concepts, but a "good environment and support" may actually constitute the availability of well-defined organization practices and supporting tools. Principle six identifies face-to-face communication as the best communication method. Face-to-face communication is an excellent way of motivating the troops, but written comprehensive details are necessary to conduct productization work within the NWC. In theory, principle six advocates good human interaction, but this is not necessarily a sufficient condition for software development. In fact, face-to-face communication has been documented in research as being frequently misunderstood, in particular when the communication is frequently evolved as happens when the communication is repeated over a period of time.

Principle eight proposes that agile processes promote sustainable development and asserts that sponsors, developers, and users should be able to maintain a constant pace indefinitely. In practice, this principle is applied to mean developers should be able to work reasonable hours in a sustained fashion to obtain the required objectives. However, this does not necessarily mean products, product-lines, and multiple domains of products can actually be sustained – which is part of a sustained development concept. Whether these principles are actually true or possible, they definitely can be barriers for use of agile methods for weapon/weapon-related projects in the NWC. Most existing NWC products, including software, have a limited development life, a much longer stockpile life, with a very limited concept of sustaining support activities by the same development personnel.

However, NWC products, the development process, and the personnel involved would greatly benefit from having a better sustainment/support concept, where every project/product was not conceived as a totally new activity. This would reduce the barriers toward application of these Agile Method principles – although the implementation mechanisms would be somewhat different than conceived by the Agile Methods approach. The sustainment/support concept would allow developers to transition from project to project, incorporate existing product designs/implementations more easily into new/updated products – since documentation would be available and developed for supportability, and integrate new personnel as needed. This would probably not result in reducing certain effort spikes in projects, but might reduce individual effort spikes by better planning and resource management across projects.

Principle nine indicates that continuous attention to technical excellence and good design enhances agility, and Principle ten identifies Simplicity - the art of maximizing the amount of work not done, as essential. Principle twelve emphasizes the importance that at regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly. All these principles are clearly applicable to weapon/weapon-related products as well as most any type of project. These principles do not represent barriers for the use of agile methods, unless some of the terms used are misinterpreted. For example, principle ten clearly means within the context of principle nine.

Principle eleven implies that the best architectures, requirements and designs emerge from self-organizing teams. Although there is some doubt that this is always true, clearly some of the best innovations have come from such teams. However, for weapon/weapon-related software projects, the teams are formed from across various organizations and sites. These teams are not likely to be self-forming, but require specific direction and requirements to get started. Advanced development teams and research teams may be more aligned with being "self-organizing". This is not a unique barrier for the NWC and would likely be found in companies of any significant size with multiple application domains and/or high consequence application products.

Although aspects of agile methods may be utilized by the NWC (both for hardware and software weapon/weapon-related projects), none of the agile method instances described in Section 2 directly address all practices that might be needed for development of software for critical systems. Therefore, none of these agile methods would be able to be followed in their entirety for a weapon or weapon-related project, including software development, maintenance, or acquisition. There are too many barriers that would prevent this from occurring.

Perhaps one of the most significant barriers to application of a specific agile method instance is the lack of a formal standard for what constitutes an "agile method". Persons and organizations who claim to be using "agile methods" may be picking and choosing those Agile Manifesto values, principles, and specific method characteristics that are most appealing for the moment, rather than some set that would by consensus be agreed to as "agile". The completion of the draft IEEE 1648, "Recommended Practice for the Introduction and Use of Agile Software Development Methods," may eliminate some of this barrier.

4.2 Summary of Conclusions and Recommendations

Agile methods for software development are based on the Agile Manifesto with the main characteristics that these methods have in common being:

- Small collocated teams,
- Collocated users/experts,
- Short, iterative, incremental development cycles,
- Experienced and knowledgeable developers.

Weapon/weapon-related software is required to be produced in an extremely rigorous manner with constraints and requirements that are frequently at odds with the principles and characteristics of the Agile Manifesto. For example, the TBP-306, which provides specific requirements and guidance for software product processes, calls out the requirements for specific life cycle evidence that provides the necessary confidence that the as-built product satisfies its requirements for specific application use and has been designed, implemented, tested, and produced using appropriate processes. Some of this life cycle evidence is required at various times during the life cycle activities. It is required that documented evidence of weapon/weapon-related software is produced for product acceptance by NNSA personnel. That software evidence must exist and face-to-face conversation is not the best communication method in this instance.

Of the agile method instances reviewed in Section 2 of this white paper, the most applicable for weapon/weapon-related use appear to be Crystal and Dynamic Systems Development Method, with the caveat that these methods must be tailored and specialized practices added for use in critical systems and weapon/weapon-related software projects. Other agile methods have some reasonable applicability to research and prototype or concept development phases of weapon/weapon-related software, but cannot be used in their entirety for weapon or weapon-related production software.

In general, there are several conclusions that can be made based on the review information presented in Section 2 and the weapon/weapon-related software attributes and requirements presented in Section 3.

(1) Conclusion 1: most agile methods reviewed provide more of a philosophical and project management approach than a software development approach; with only a few exceptions, these approaches have some applicability to weapon/weapon-related software development and/or support

(2) Conclusion 2: there is significant variation in the capabilities provided by the various agile methods; this supports the need for a standard such as IEEE 1648 to provide better guidance as to what methods can legitimately be called “agile”

(3) Conclusion 3: none of the agile methods directly address practices that might be needed for development of software for critical applications, although such practices might be integrated as needed

(4) Conclusion 4: the agile method practices that are most non-applicable to weapon/weapon-related software are the same ones that support certain aspects of the principles described in Section 3.3.2.1.2 that are either limited or non-applicable: P1 and P3 – applied to frequent delivery of working software; P4 –

applied to co-location of customer and project team; P6andP7 – applied to face to face communication and working software (without supporting written documentation with configuration control as evidence of iteration and completion)

(5) Conclusion 5: all agile methods reviewed appear to require (or at least need) motivated, creative, and talented software developers for success; it might be reasoned that all projects would be more successful with this baseline requirement

(6) Conclusion 6: of the agile methods reviewed, the most applicable for weapon/weapon-related use appear to be Crystal and Dynamic Systems Development Method; other methods have more limited application

(7) Conclusion 7: the agile methods reviewed all appear to have some reasonable applicability to research and prototype/concept development phases of weapon/weapon-related software

In addition, it is recommended that this white paper be shared with the IEEE 1648 standard-development team that has so graciously shared their early draft proposals with the contributors to this white paper. It is also recommended that this information be shared with each site through the SQAS contributors, and as possible, shared through professional conferences and publications as the opportunity may arise.

Appendix A: Resources

A.1. References

TBP-PRP, "Product Realization Process," Technical Business Practice, Nuclear Weapons Complex

TBP-306, "Software Product Processes," Technical Business Practice, Nuclear Weapons Complex.

DG10235, "Software Product Processes," Design Guide, Nuclear Weapons Complex.

Agile Manifesto, 2001, **http://www.agilemanifesto.org/**

CMMI, "Capability Maturity Model Integration," Version 1.1, CMU/SEI-2002-TR-028(continuous) and CMU/SEI-2002-TR-029 (staged), Software Engineering Institute.

http://c2.com/cgi/wiki?CapabilityMaturityModel

Krasner, Herbert, "Agility and Quality (or) What CMM Level is my XP Project?" **http://lonestar.rcclub.org/ASEE/asee-talk.pdf**

"Agile Development and the CMMI:

Ornburn, Steve, Kane, David, "Anti-Matter and Matter or Reconcilable Differences?" **http://www.sstc-online.org/proceedings/2002/SpkrPDFS/WedTracs/p752.pdf**

Ambler, Scott, Agile Modeling and eXtreme Programming (XP) **www.agilemodeling.com**

Radding, Alan, "Extremely Agile Programming: Agile Programming Systems" **http://www.computerworld.com/softwaretopics/software/appdev/story/0,10801,67950,00.html**

Pettichord, Bret, "Testers Should Embrace Agile Programming" **http://www.io.com/~wazmo/papers/embrace_agile_programming.html**

Clements, Paul, Ivers, James, Little, Reed Nord, Robert Stafford, Judith,"Documenting Software Architectures in an Agile World" **http://www.sei.cmu.edu/publications/documents/03.reports/03tn023.html**

IEEE 1648, "Recommended Practice for the Introduction and Use of Agile Software Development Methods," draft, April 2006.

Abrahamsson, P., Salo, O., Ronkainen, J., Warsta, J., "Agile Software Development Methods: Review and Analysis," VRR Publications 478, Espoo 2002.

A.2. Definitions

Agile Methods: a discipline of developing software where the practices adhere to the Agile Manifesto Values and its Principles

Agile Manifesto Values: In summary, from the Agile Manifesto reference. Agile software development is based on a hierarchy of values.

1. Individuals and interactions are valued over processes and tools.
2. Working software is valued over comprehensive documentation.
3. Customer collaboration is valued over contract negotiation.
4. Responding to change is valued over following a plan.

Agile Manifesto Principles: In summary, from the Agile Manifesto reference. Agile software development is based on the following principles that provide additional context for the values:

1. Customer is highest priority – early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development.
3. Deliver working software frequently.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals, good environment and support, trust.
6. Face-to-face conversation is the best communication method.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity--the art of maximizing the amount of work not done--is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Weapon Material: NNSA nuclear weapons, assemblies, components, software, or parts thereof.

Weapon-Related Material: any material, including associated software, test and handling equipment, tooling, and fixtures, being developed and produced for or by the NNSA and intended for use in conjunction with, or in any way related to, weapon development, engineering, production, surveillance, or dismantlement.

Weapon/Weapon-Related Software Product: software that is weapon/weapon-related material, developed in accordance with the Nuclear Weapons Complex (NWC) Technical Business Practices (TBPs), in particular TBP-306.

A.3. Acronyms

ASD	Adaptive Software Development
BPR	Business Process Re-Engineering
CMMI	Capability Maturity Model Integrated
DoD	Department of Defense
DOE	Department of energy
DSDM	Dynamic Systems Development Method
FDD	Feature Driven Development
ICO	Interagency Contractor Order
IMS	Image Management System
ISO	International Organization for Standardization
LD	Lean Development
MoSCoW	Must have o Should have o Could have o Won't have
NNSA	National Nuclear Security Administration
NWC	Nuclear Weapons Complex
OTS	Off-The-Shelf
PLD	Programmable Logic Device
PROM	Programmable Read Only Memory
PRP	Product Realization Process
PRT	Product Realization Team
QA	Quality Assurance
QADR	Quality Assurance Defect Report
QAIP	Quality Assurance Inspection Procedure
SQAS	Software Quality Assurance Subcommittee
TBP	Technical Business Practice
TQM	Total Quality Management
WBS	Work Breakdown Schedule
XP	eXtreme Programming