Fundamentals of Agile Systems Engineering – Part 2

Rick Dove
Paradigm Shift International
Taos County, New Mexico, USA
dove@parshift.com

Ralph LaBarge
Johns Hopkins University/APL
Laurel, Maryland, USA
ralph.labarge@jhuapl.edu

Revision 20-May-2018 made corrections and updates, changing UURV framework to CURVE, and modules to resources.

Abstract

Agile systems-engineering and agile-systems engineering are two different concepts that share the word agile. In the first case the system of interest is an engineering process, and in the second case the system of interest is what is produced by an engineering process. The word agile refers to the adaptability and the sustainment of adaptability in both types of systems. Sustained adaptability is enabled by an architectural pattern and a set of system design principles that are fundamental and common to both types of systems. Research that identified this architectural pattern and design principles is reported, updated, and applied here in two Parts. Part 1 focuses on agile-systems engineering, reviewing the origins, values, and core concepts that define and enable domain independent agility in any type of system. Part 2 focuses on agile systems-engineering, identifying core agility-enabling concepts in the software-development domain-specific practice known as Scrum, reviewing an agile hardware/software satellite-development systems-engineering case for its source of agility, and then suggesting the development of an agile systems-engineering life cycle model as a natural next step.

Introduction

An agile systems-engineering process is itself a system, gaining its agility from a fundamental architecture and set of design principles that enables adaptability. This architecture and set of design principles were the subject of a companion article of the same name, but designated as Part 1 (Dove and LaBarge 2014), with its focus on agile-systems engineering. A brief review of Part 1 will set the stage for applying the architecture and design principles to an agile systems-engineering process, the focus of this Part 2 continuation.

Agile-Systems Background

Agility is defined as the ability of a system to thrive in an uncertain and unpredictably evolving environment; deploying effective response to both opportunity and threat, within mission. Effective response has four metrics: timely (fast enough to deliver value), affordable (at a cost that can be repeated as often as necessary), predictable (can be counted on to meet the need), and comprehensive (anything and everything within the system mission boundary). This is a core definition of agility exhibited by an agile systems engineering process, as well as an agile system developed by any process, agile or not.

Explained in Part 1, agile systems are designed in counterpoint to their operating environments, characterized as a Capricious, Uncertain, Risky, Variable, Evolving (CURVE) framework:

- Capricious: randomness among unknowable possibilities.
- Uncertain: randomness among known possibilities with unknowable probabilities.
- Risky: randomness among known possibilities with knowable probabilities.
- Variable: randomness among knowable variables and knowable variance ranges.
- Evolving: gradual (relatively) successive developments.

Research that began in 1991, originating the concept of agile systems and putting the word *agile* into play, examined a large number of diverse systems throughout the '90s that exhibited agile capabilities. This research abstracted core architecture and design principles that appear necessary and sufficient to enable agility in systems and processes of any kind (Dove 2001). The architecture will be recognized in a simple sense as drag-and-drop, plug-and-play, loosely coupled encapsulated modularity, but with some critical aspects generally ignored in descriptions of modular architecture.

Agile systems are designed for change. They can be augmented with new functional capability. They can be restructured with different internal relationships among their constituent systems or subsystems. They can be scaled up or down for economic delivery of functional capability. They can be reshaped to regain compatibility or synergy with an environment that has changed shape. These types of changes are structural as well as functional, and require an architecture that accommodates structural change, whether the system of interest is a development process or the product of a development process.

There are three critical elements in the agile architectural pattern: a roster of drag-and-drop encapsulated resources, a passive infrastructure of minimal but sufficient rules and standards that enable and constrain plug-and-play interconnection, and an active infrastructure that designates four specific responsibilities for sustaining agile operational capability: resource evolution, resource readiness, system configuration, and infrastructure evolution.

Agile Systems-Engineering Context

Systems engineering is a disciplined activity that delivers engineered solutions to problems and opportunities – an activity often involving multiple stakeholders, coordination across multiple engineering disciplines, and complexity in both problem and solution (Sheard 2000). Unlike other engineering disciplines that employ natural laws to guide and govern engineering design with certainty within a single discipline, systems engineering deals with the social, political, and technical aspects of managing projects that span multiple disciplines. These projects can be quite large and complex, need cross-discipline unifying architectures and operational concepts, require multi-party accommodations to resolve tradeoffs, and often exhibit unexpected emergent behaviors as the project progresses.

Peter Checkland, in speaking of "hard" and "soft" systems thinking (Checkland and Holwell 2004, 45-46), characterizes the "hard" variety as classic systems engineering "appropriate in well-defined problem situations in which well-defined objectives are accepted and the live issues concern how best to engineer a system to meet them. … On the other hand, 'soft' approaches are said to be appropriate in messy problem situations, characterized by obscure objectives and multiple clashing viewpoints." Checkland's Soft Systems Methodology (SSM), designed to learn and find effective responses to real world changing problem environments, has valuable and practical application in an agile systems-engineering life cycle model; but that pursuit is outside the scope of this current article. Much of SSM, however, is the (unrecognized) foundation of the agile software development management process known as Scrum, which will be explored later.

In agile systems-engineering, defining a solution in terms of the problem understanding and defining the problem in terms of the solution understanding is a spiral iteration of discovery and learning convergence through an evolving engineering environment. If the operational environment for the deployed system is also evolving, systems development continues beyond the development and production stages all the way to retirement, and benefits if the system produced is itself an agile system that facilitates continued change and adaptation.

Illustrated in Figure 1, the separate environments of both the engineering system and the engineered system define the needs and degrees for agility in each of those systems. If the engineering environment is not stable and predictable, an agile approach is appropriate; which will evolve engineered system variations, creating a dynamic requirements environment for the engineered system, at least during development. There is an interaction between both of these systems during development, as the engineered system matures in modeling, simulation, and prototype instantiations that provide feedback analysis to the engineering system. That process should continue throughout the remainder of the engineered system's life cycle if it is deployed in a dynamic evolving environment. Of course, the engineered system's life cycle may be cut disappointingly short if it is not designed and sustained as an agile system.

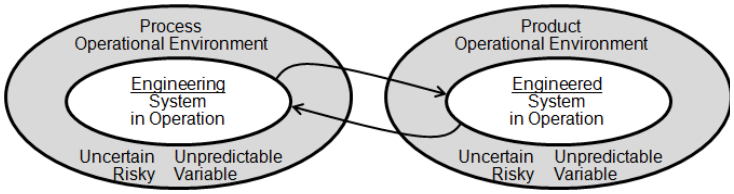

Figure 1: Two different operational environments defining necessary agile counterpoint for the systems they encompass.

Agile Systems-Engineering and the System Life Cycle

Here the focus is on domain independent agile systems-engineering, a type of systems engineering distinguished by its ability to deliver functional system-engineering value in a dynamic systems-engineering environment.

Life cycle, as a term applied to systems, traditionally demarcates the progressive maturity flow of a system through a linear sequence of stages, from concept to disposal. Inherent in this model are the notions that a system is in one and only one stage at any point in time, and progresses from one single-state stage to another in a proscribed sequence.

For practical purposes one cannot argue with the notion that there are times when a system does not exist that bound a time when a system does exist. Nor can one argue against the necessity to develop a system before utilization can occur. Here, however, the argument is against the continued notions of non repeating stages and of single-state existence by depicting an agile life cycle in Figure 2 as having progressively concurrent repeated stages.

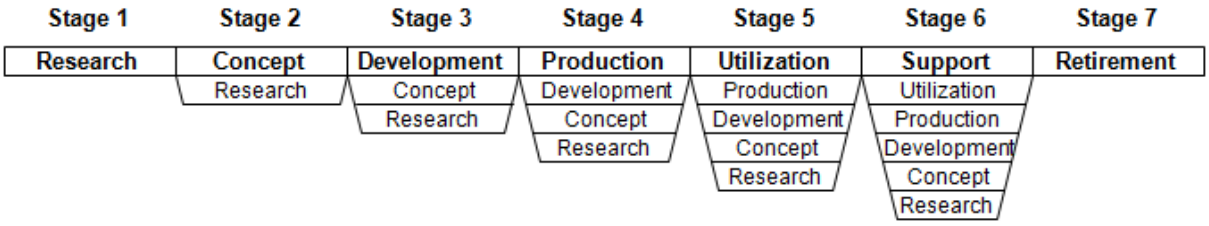

Figure 2: Framework of agile system engineering life cycle model, depicts constant evolution of all prior ISO/IEC 15288 life cycle stages as the life cycle progresses through maturity

ISO/IEC 15288 describes “the” generic system life cycle as a seven stage maturity process progressing through: exploratory research, concept, development, production, utilization, support, and retirement (ISO/IEC 2008, 10-14). This is a classic waterfall model at the macro level, with 15288 recognizing that each stage can be further reduced into maturity flows as well, which may or may not be waterfall models.

In reality, the 15288 stages of utilization and support are typically concurrent. Also, in reality, all stages are not necessarily true conditions of the system, but rather perceptions and/or claims of an individual or collective authority.

Looking at just two of the stages reveals the issue: must a system be either in a state of development or a state of utilization, but not both at the same time? A resilient system exists by definition in this dual state. So do agile systems, self organizing systems, autonomic systems and other such dynamic systems that exist sustainably in an uncertain and unpredictable environment. Think of a city as a system, and consider that the subsystems of that city are many, are individually in a full variety of the different stages, and each typically moves among all of the different stages repeatedly over time.

Figure 2 shows a sequential system engineering maturity transition across the primary stages, but also recognizes that within each of the sequential primary stages is a growing concurrency of all of the prior stages. Agile systems engineering processes, by definition, are capable of responding to their environment as their environment changes, regardless of why or when these changes occur. Note that an agile systems engineering process has difficulty exercising agile capabilities if it does not develop an agile system, one that has an architecture which facilitates justified change throughout the development and subsequent utilization and support stages.

Each of the seven stages of the systems-engineering life cycle may have individually different operational environments, ranging from stable to unpredictable. Dealing with stage-specific agile needs and methods is beyond the scope of the current article.

In the domain context of agile software development, the production stage is the first build/release stage that puts product into the operational user environment, initially – not as a temporary prototype, but as a usable working product. Subsequent increments and iterations of development occur during utilization and support of prior releases.

An effective agile systems engineering process must converge on sufficient completion of each of the primary stages to warrant the transition to the next primary stage, presumably on schedule and on budget regardless of how flexible or rigid these might be. This is represented in the Figure 2 life cycle depiction showing diminishing emphasis on the lower concurrent stages as maturity through primary stages progresses.

The software-development management process known as Scrum is examined next; both because it is strongly associated with agile software development, and because it has excellent concepts that can be adapted to domain independent agile systems engineering, if these concepts are understood for how and what they contribute to agility.

Baseline Scrum

Scrum is one of the most popular project management practices for agile software development, and is highly associated with the concept of agile systems-engineering and the word agile in many people’s minds unfamiliar with agile concepts outside of the software domain. Consequently, the classic Scrum process will be examined as described in (Schwaber and Sutherland 2013) as a base

line for what makes it a truly agile process in the software development domain. The reader familiar with agile software development practices is advised that neither the Agile Manifesto's four concepts nor its 12 backup principles are responsible for the agility of the Scrum process, as will be shown. These concepts and principles are instead among well known (but not necessarily practiced) best project management practices and human productivity amplifiers, regardless of whether an agile systems-engineering (or agile software development) process is called for.

Overview of Classic-Scrum

In deference to readers that may not be versed in the Scrum process, as well as those that are unfamiliar with The Scrum Guide (Schwaber and Sutherland 2013), key elements are reviewed here, with quoted remarks in this section taken from that guide.

"Scrum (n): A framework within which people can address complex adaptive problems, while productively and creatively delivering products of the highest possible value."

"Scrum is founded on empirical process control theory, or empiricism. Empiricism asserts that knowledge comes from experience and making decisions based on what is known. Scrum employs an iterative, incremental approach to optimize predictability and control risk. Three pillars uphold every implementation of empirical process control: transparency, inspection, and adaptation." Scrum is an engineering process that works toward a solution in a series of steps, questioning the progress and process at the completion of each step. This requires complete transparency of everything that has been decided and accomplished, objective assessment of the quality and value of what has been accomplished and how it has been accomplished, and continuous feedback learning that adjusts both process and product to minimize variance from goals.

"Scrum's roles, artifacts, events, and rules are immutable and although implementing only parts of Scrum is possible, the result is not Scrum. Scrum exists only in its entirety and functions well as a container for other techniques, methodologies, and practices." In the systems engineering perspective, Scrum is a management process architecture that can accommodate a wide variety of technical processes.

"Scrum Teams deliver products iteratively and incrementally, maximizing opportunities for feedback. Incremental deliveries of 'Done' product ensure a potentially useful version of working product is always available." In Scrum an increment is called a *sprint,* and multiple sprints are construed as iterations on increments. Unlike Alistair Cockburn's (Cockburn 2008) well written distinction between increments and iterations, Scrum iterations typically add new features as well as improve the performance of existing features.

"The heart of Scrum is a Sprint, a time-box of one month or less during which a 'Done', useable, and potentially releasable product Increment is created. Sprints have consistent durations throughout a development effort. A new Sprint starts immediately after the conclusion of the previous Sprint." Note that Sprints have consistent time durations, which establishes a cadence to the total project; and a Sprint ends when the allotted time duration expires, whether or not all work planned for the Sprint is completed. This hard stop is intended to improve the task and Sprint estimation capabilities of the developers. Unfinished work can be picked up in a subsequent Sprint.

"Scrum is a framework for a project whose horizon is no more than one month long, where there is enough complexity that a longer horizon is too risky. The predictability of the project has to be controlled at least each month, and the risk that the project may go out of control or become unpredictable is contained at least each month." The time horizon resolves uncertain requirements

with a series of small investments in experimental development, evaluation learning, and adaptive correction if necessary. The allowable time horizon is a measure of the estimated uncertainty appropriate for a Scrum approach. The intent is to affordably contain the costs of incremental learning and correction.

"Scrum users must frequently inspect Scrum artifacts and progress toward a goal to detect undesirable variances. … If an inspector determines that one or more aspects of a process deviate outside acceptable limits, and that the resulting product will be unacceptable, the process or the material being processed must be adjusted. An adjustment must be made as soon as possible to minimize further deviation. Scrum prescribes four formal opportunities for inspection and adaptation." These are called the Sprint Planning Meeting, the Daily Scrum, the Sprint Review, and the Sprint Retrospective. All four are collaborative learning events with all team members present and participating.

"Optimal Development Team size is small enough to remain nimble and large enough to complete significant work. Fewer than three Development Team members decreases interaction and results in smaller productivity gains. … Having more than nine members requires too much coordination. Large Development Teams generate too much complexity for an empirical process [learn by doing and evaluating] to manage." This suggests the sweet spot for participative collaborative-learning diversity that spurs productivity lies between three and nine. Larger projects are then composed of multiple Scrum teams often organized as a Scrum of Scrums.

Shown in Figure 3, Scrum includes three major roles, four formal types of meetings, and two formal project management artifacts in an incremental and iterative framework (Sutherland and Schwaber 2007).

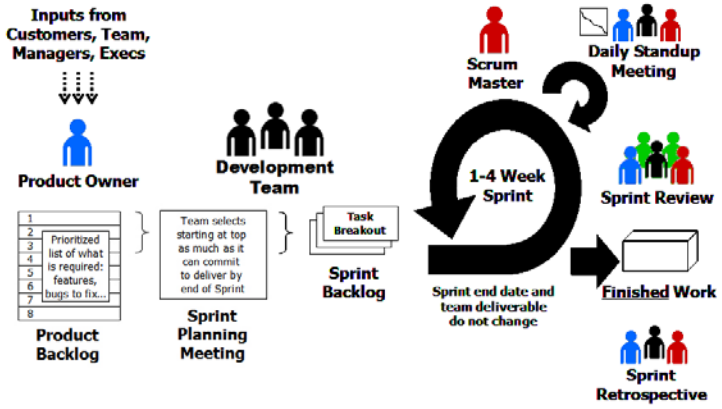

Figure 3. Scrum (from Sutherland and Schwaber 2007)

The three major Scrum roles are the Product Owner, the Scrum Master, and the Development Team. The Product Owner is responsible for defining, communicating, and prioritizing the product development tasks; and for accepting or rejecting the incremental product delivery at the end of each Sprint. The Scrum Master is responsible for ensuring that the Scrum process is understood, and that the practice adheres to Scrum theory and rules; and will also train and coach the Scrum process. The Development Team is a self-organized, cross-functional group that performs the product design, development, integration, test, and demonstration tasks for each Sprint; and they are solely responsible for choosing how to accomplish these tasks.

The four formal meetings are Sprint Planning, Daily Scrum, Sprint Review, and Sprint Retrospective. With the exception of a fixed 15-minute Daily Scrum, these meetings are time-boxed relative to the duration of a Sprint, with times that follow shown for a 4 week Sprint. The Sprint Planning meeting timed at eight hours has two roughly equal parts: 1) the Product Owner

explains the tasks in the Product Backlog and their priorities, from which the Development Team decides how many of the high priority tasks to move into the Sprint Backlog, and 2) the Development Team then collaborates to agree amongst themselves on how they will complete the tasks in the Sprint Backlog during the next Sprint. The Daily Scrum is a fixed 15-minute meeting designed to quickly assess and resolve the state of the current Sprint as each Development Team member answers three critical questions: what did I accomplish yesterday, what am I planning to accomplish today, and what impediments are in my way? The Sprint Review meeting timed at four hours occurs at the end of the Sprint in two roughly equal parts: 1) the incremental product is demonstrated to the Product Owner, and 2) there is a discussion of positive and negative development-related lessons learned that can help subsequent Sprints. The Sprint Retrospective is timed at three hours and occurs after the Sprint Review and before the next Sprint Planning meeting, and examines how the Sprint went relative to people, relationships, process, and tools; identifying what went well and what needs improvement; and creating a plan for improving how the team does its work. With the exception of the Sprint Planning meeting, Scrum meetings are assessment, learning, and adaptation focused.

The two formal project management artifacts are the Product Backlog and the Sprint Backlog. The Product Backlog is a prioritized list of requirements, expressed as tasks that can each be implemented within the time-box limitations of a single Sprint; with complex tasks broken down into a series of sub-tasks that can be incrementally completed over several Sprints. The Sprint Backlog is a collection of specific design, development and test activities allocated to the current Sprint in process or about to start. Other less formal artifacts are typically used to monitor progress during a Sprint, such as the so-called Burndown Chart, which displays daily updates of the estimated remaining work hours required to complete the Sprint.

Interpretation of Classic-Scrum

Scrum is a discipline based on feedback learning, designed to remove uncertainty and risk about the system to be engineered, and designed to increase productivity of the engineering process. The learning occurs in fixed-schedule periodic assessments and adaptations, with feedback functioning much like a thermostatic control system uses negative feedback to reduce temperature variance from a set point. This feedback learning is intended to evolve both the engineered product and the engineering process in successive Sprints, with fitness feedback at Sprint Reviews and Sprint Retrospectives, much like natural selection evolves a species through successive generations. The value of this approach is completely dependent upon the quality of the Review and Retrospective objectivity and scope. The Scrum Master is responsible for the level of achieved quality or lack thereof in this feedback learning approach. Going through the motions with marginal benefit has little if any value.

Evolutionary learning happens between Sprints, real-time learning happens during Sprints. Real-time learning is the function of the 15-minute Daily Scrum meeting. Every participant in this meeting provides personal status in three areas: what I did yesterday, what I'll do today, and what impedes my progress. The value here is realized in what is heard and processed by others. Going through the motions without active collective listening provides no value. Again, it is the Scrum Master's responsibility to ensure this meeting's effectiveness.

Scrum raises the team's collective system-wide awareness and reveals unpredictable emergent behaviors, potentially revealing undesirable emergent futures before they occur.

The explicit essence of successful Scrum is *effortful* learning through active collaborative communication. Effortful learning is a self-motivated process that continuously identifies the next thing to learn after successfully accomplishing the last learning objective. In the case of a Scrum team, effortful learning is a key focus and responsibility of a competent Scrum Master.

The implicit essence of successful Scrum, however, is the ability to effectively adapt the process and the product to what has been learned. This means changing what is being done in product development and changing how it is being done in the team's working process – which requires an agile (adaptable) architecture of both product and process to be effective. Though absent from the explicit principles and processes of Scrum, ensuring that agile architectures underlie both product and process are key responsibilities of the Product Owner.

You don't hear the dependence of successful Scrum on agile architectures explicitly called out in the Scrum Guide or the Scrum Papers. Perhaps, on the product side, this is because Scrum was born and lives to solve software development problems in a time when software development employs Object Oriented Programming (OOP) techniques, which inherently provide the basic necessary structural needs for system adaptability – drag-and-drop modularity in a plug-and-play infrastructure. Thus, replacing, augmenting, or reconfiguring elements of software systems in successive Sprints is facilitated as learning drives adaptation. As will be shown in the next section, hardware product systems can also facilitate adaptation of Scrum-learning with an agile product architecture.

On the process side, Scrum theory and practice do not explicitly address the active infrastructure process management responsibilities necessary to sustain agility in the process architecture. Figure 4 assigns the architecture's four active-infrastructure responsibilities to appropriate parties employed in Scrum. The process resources are principally the people. The competence quality of

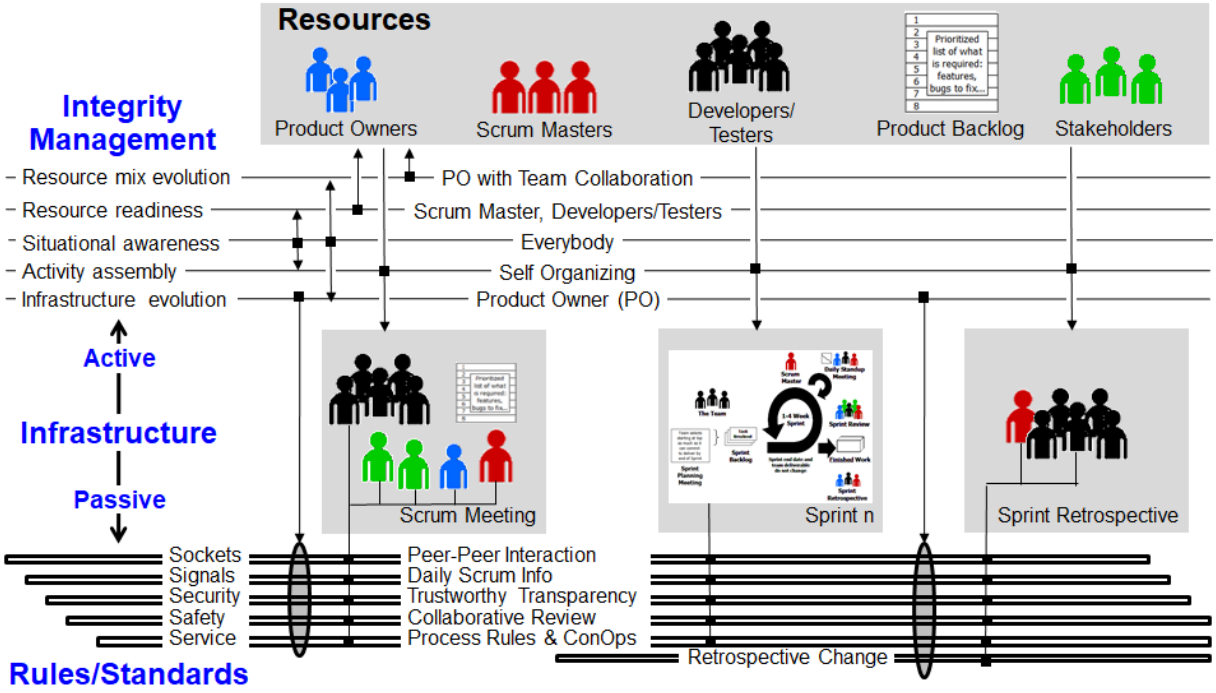

Figure 4. The Scrum Agile Architecture Pattern

Product Owners, Scrum Masters, and Developers are critical for obtaining and sustaining the benefits of the Scrum process. The Product Owner in this depiction is charged with the responsibility of staffing, assessing, re-balancing, and evolving the human components in the Scrum system The Scrum Master in this depiction appears as a resource, appears in the active infrastructure as the system assembler, and appears in the passive infrastructure as the enforcer of plug-and-play resource interconnectivity standards.

Scrum is a proactive initiative, productively confrontational in nature. No denial is possible when Scrum is done right. It is highly intense in the people interaction area. The Scrum Master is the process coach and enforcer, as well as the "servant leader" enabling team productivity.

When is Scrum, or something like it, the right approach? When the team has little experience working together, when the problem and solution pair are insufficiently understood, when the emergent behavior of interaction complexity can produce unpredictable costly results, when the system and/or process environment are likely to evolve unpredictably, when stakeholder commitment to budget and schedule are uncertain. "Scrum is a framework for a project whose horizon is no more than one month long, where there is enough complexity that a longer horizon is too risky. The predictability of the project has to be controlled at least each month, and the risk that the project may go out of control or become unpredictable is contained at least each month."

What if the Scrum Master and Product Owner are not what is required to reap the Scrum benefits? In such cases this contributes to dissatisfaction with the process among the participants, and with disappoint with the process among management. Experience has shown that many adoptions of the Scrum process are dissatisfying disappointments (Sutherland 2007, 79). However, in some cases Scrum provides a management framework that can be better than what it replaces even if it doesn't deliver much of the Scrum learning and adaptation potential. In these later cases it offers prescriptive shared procedure where perhaps none was present before.

An Agile Systems-Engineering Example

The CubeSat program introduced in Part 1 defines an agile-system architecture for developing and constructing CubeSats from a combination of community-available common-off-the-shelf components and internally-developed mission-specific components. The CubeSat agile-system architecture specifies internal plug-and-play infrastructure for satellite construction and external interfaces for launch-vehicle compatibility. The internal infrastructure specification enables agile mission-specific designs that can avoid the cost of architecture and infrastructure design, take advantage of widely available COTS resources, and focus development on mission-specific resources. The external interface specification ensures that any CubeSat can be deployed by any member of a conforming launch-vehicle family, providing agility at the higher mission-deployment system level as well.

CubeSat specifications say nothing about the system-engineering process that will develop and assemble a mission specific CubeSat at any of the many organizations that do these projects. Given the financial and window-of-opportunity risks associated with terminal deployment of a system that cannot be returned for replacement, correction, or design update, a common systems-engineering process would likely be some variation of a traditional waterfall engineering process, with big upfront planning and design.

A group at Johns Hopkins University Applied Physics Laboratory (JHU/APL) thought differently. Maybe because they had to. Quoted statements in this section, if not otherwise credited, are from

(Huang et al, 2012). “The Multi-Mission Bus Demonstration (MBD) is a Johns Hopkins University Applied Physics Laboratory program to demonstrate a government sponsored mission in the standardized 3U (10 x 10 x 30 cm) CubeSat form factor. … With vicious cost and schedule control, the MBD project is providing a classified DoD payload that will revolutionize the mission area and provide an operationally relevant capability to the war fighter. … The MBD space vehicles will cater to mission operation versatility and rapid response launch capabilities.”

“MBD is an Advanced Concept Technology Demonstration. To complete the demonstration, two ready-to-launch spacecraft based on non-proven technology had to be designed and developed in less than 10 months and under $10 million dollars. With little or no COTS parts qualified to meet the mission requirements, new hardware and software development was required. The MBD project is characterized as a super-high-tech project; i.e., new, non-proven concepts requiring extensive development of technologies and system components.”

NASA reliability requirements encourage the reuse of heritage hardware, proven in the past 1,000 satellite development efforts. But the unique CubeSat form factor, at this point, had only 70 Missions. Compatible heritage hardware was not available from the CubeSat COTS community. The MDB mission requirements called for innovative technologies far advanced over anything done previously. “The small volume of a CubeSat platform remains a daunting engineering challenge to overcome. Components were not ‘plug and play,’ requiring, in some cases, vendor collaboration and modification to meet the requirements of the MBD program.”

JHU/APL has more than 70 years’ experience in the design, build, and operation of over 60 spacecraft and 200 instruments, using a tight requirements process and disciplined development to meet NASA space flight requirements. It was evident to the MDB program team that their proven systems engineering procedures would be unable to meet cost and schedule in this highly uncertain technical development effort. They’d have to do something very different to reveal and reduce the uncertainties rapidly and cost effectively.

They did. Their paper, *Agile hardware and software systems engineering for critical military space applications* (Huang et al. 2012), “…discloses the adaptation and application of concepts developed in agile software engineering to hardware product and system development for critical military applications … created a large paradigm shift from traditional spacecraft development.”

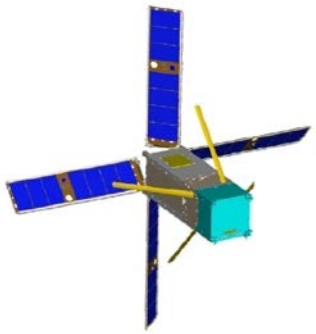

Figure 5 – JHU/APL MBD CubeSat with deployed solar array.

Project uncertainty was rooted in the combination of a small physical envelope constraint, high technical capability requirements, an unprecedented low budget, and an unprecedented short program duration. This was recognized by the MBD sponsor, who was willing to make compromises and accept more risks than the typical NASA mission “in order to balance cost, schedule, and reliability while still meeting all mission requirements. To meet the dramatically constrained volume, costs and schedule while increasing functionality more than ever seen in a CubeSat format, new designs and concepts needed to be created, developed, and manufactured. … The MBD spacecraft [shown in Figure 5] is designed with all the complex and critical subsystems found within a typical earth observing multi-instrument satellite.”

Taking stock of the project environment, unpredictability and uncertainty appeared high, and a

real risk of project success existed. Couched in the CURVE framework outlined in Part 1:

- Capricious: Appetite for stakeholders to stay the course when things look uncoordinated, or when unresolved development issues are allowed to persist. Cultural adjustment of engineers working outside their standard procedures.
- Uncertain: what requirements to use as technical drivers, what technical path to take, how changed subsystem dependencies will interrupt momentum, what untraditional decisions will have to be made, what SME expertise will be needed.
- Risky: it can't be, or doesn't get, done within the constraints.
- Variable: nothing relevant foreseen.
- Evolving: nothing relevant foreseen.

A Scrum-Like Approach to Agile Systems-Engineering

JHU/APL utilized a flat organization structure with six distinct development teams including Payload, Electrical, Software, Mechanical, Ground & Navigation Control, and Avionics. The lead of each development team, called a "Deputy", reported directly to the MBD Program Manager, called the "Sherriff". Each development team lead also had direct access to the Customer. The Program Manager reported directly to the head of the JHU/APL Space Department and had the authority to implement any technical or programmatic changes required to ensure the success of the MBD program. Each development team included a small interdisciplinary group of designers and developers, called a "Posse". While the names for each role are different, the MBD program essentially used Scrum as the basis for the organizational structure, with the Sherriff serving as the Product Owner. Each Posse was essentially a parallel scrum Development Team led by a Deputy, who acted as the team's Scrum Master.

At the start of each day the MBD team held a "Round-Up" meeting which was attended by everyone on the program. These daily meetings were used to review issues and priorities, and to allow each team to solve problems and react to changes quickly. All of the JHU/APL team leads (Deputies) were collocated in an open office to assure easy and frequent communications between the parallel Development Teams. The MBD program used a traditional Scrum Board to track tasks, issues and assigned priority levels, to identify which team member was responsible for the completion of a task, when that task was scheduled for completion, and the dependencies between tasks.

The emphasis for each Development Team was to deliver working components as early as possible in the development cycle. Rather than hold classic waterfall style life cycle reviews such as a System Requirements Review, Preliminary Design Review, and Critical Design Review, the MBD team held a single comprehensive design review, called the One Design Review, but also held many informal peer reviews with subject matter experts throughout the program life cycle. Whenever possible off-the-shelf components were used. When custom hardware items were required they were designed and manufactured in house using JHU/APL high precision manufacturing facilities. Resource, subsystem, and system level testing was performed using a "build a little, test a little, learn a lot" framework designed to find and resolve issues as early as possible.

Figure 6 shows the system life cycle for the MBD program, which included both iterative and incremental development strategies. During the MBD program several different stages of the

ISO/IEC 15288 life cycle were performed in parallel. For example the Requirements & Concepts, Design & Development, Implementation & Integration, and Test & Evaluation efforts shown in Figure 6 are roughly equivalent to the 15288 Research, Concept, and Development stages. As shown in Figure 6 these four efforts were performed in parallel for a significant portion of the MBD program's life cycle.

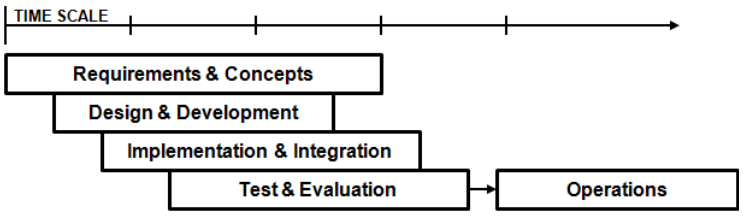

Figure 6. MBD System Life Cycle with Overlapping Stages (Huang, et al. 2012)

According to the MBD development team, using agile system engineering processes was critical to the success of the program. Specifically they said "the process flow used on MBD left the window open to make modifications at a later part of the development. By embracing uncertainty and carrying open issues forward, changes could be made without dramatically affecting other sub systems. Issues were prioritized and the development team worked to close the items that would force modifications and changes to the system as soon as possible" (Huang, et al. 2012).

The MBD program used sprints with one-day durations, while classic Scrum typically uses sprints one to four weeks in duration. Consequently, Sprint Planning, Daily Scrum, Sprint Review and Sprint Retrospective meetings were combined into a single "Round-Up" meeting at the start of each work day. The MBD Program Manager assumed the role of Product Owner. The program implemented six parallel agile development efforts for the Payload, Electrical, Software, Mechanical, Ground & Navigation Control, and Avionics subsystems of the MBD satellite. Each development team lead performed a role similar to a Scrum Master for their subsystem, but also acted as a Development Team member at the system level. Thus the MBD agile process implementation was similar to a Scrum of Scrums with a sprint length of one day. The need to design, develop and test custom hardware, such as the deployable solar array, required the MBD team to coordinate very short duration software development sprints (1 day) with longer duration hardware development sprints (1 month or more). When custom hardware was required, the MBD team built two prototypes for each hardware element. Each of the hardware prototypes was integrated into the two satellites being built during the program. To the extent possible prototype hardware items were used as production items in the final satellites, even if that required rework such as cutting printed circuit board traces and adding wires to implement a design change.

Agile Architecture and Design Principles

Agile systems and agile systems-engineering processes gain their agility from an architecture and set of design principles that facilitates sustainable adaptability, as discussed in Part1. The MBD program's agile architecture pattern is sufficiently similar to the Scrum Agile Architecture Pattern shown in Figure 4 that little of relevance would be revealed with discussion here. However, the ways in which ten design principles manifest in three categories may be instructive.

Reusable principles include Modularity, Plug Compatibility and Facilitated Reuse. The agile processes used by the MBD program's six subsystem development teams were modular in nature. Once the form, fit and functional requirements were defined for a subsystem the Development Team was asked to deliver incremental capabilities that met those requirements as early as possible in the development schedule. Changes or improvements made to the internal design or

implementation of a subsystem that did not impact the external form, fit or function of the subsystem could be made quickly, with the Development Team lead (Scrum Master) having the authority to make the required design decisions for their subsystem. Well defined and flexible interface standards were defined early on in the program so that individual subsystems could be plug compatible with the other subsystems as well as the spacecraft bus itself. Team members could be and were assigned to different subsystems at different times, provided their skill sets and experience matched the task requirements, employing the Facilitated Reuse principle in this Scrum of Scrums architecture.

Reconfigurable principles include Peer to Peer Interaction, Distributed Control and Information, Deferred Commitment, and Self-Organization. Peer to Peer Interaction was employed in the combination of Daily Roundup meetings attended by each team member as well as the colocation of staff members so that communication among team leads and team members was direct, fast and efficient. Distributed Control and Information was employed in the Daily Roundup meetings, where information was exchanged among all team members. Each team lead had the authority to make design and implementation decisions for their subsystem, while the Program Manager had the authority to make system-wide design and implementation decisions. Decisions at both the subsystem and system levels were made by the staff members who had direct access to the information required to make an informed decision as well as the ability to take immediate steps to implement the decision. The principle of Deferred Commitment was used throughout the MBD program. The MBD engineering team delayed the finalization of designs to allow many new components to be completed and fully tested. Deferring commitment to a specific design allowed modifications to requirements and hardware at a later stage of the development process. Self-Organization was employed within the subsystem teams which were self-organizing in nature. Each team lead had the flexibility to determine which tasks would be accomplished each day, and which team members would be assigned to those tasks. The use of a traditional Scrum Board assisted the self-organization process by allowing team leads and team members to quickly determine if other subsystems were dependent on the timely completion of one of their tasks.

Scalable principles (Dove and LaBarge 2014) include Evolving Infrastructure, Redundancy and Diversity, and Elastic capacity. The systems-engineering process rules and standards were established at the start of the MBD program. Evolving Infrastructure was employed in the Daily Roundup meeting discussions on what was working well and what could work better, and then be quickly implemented by the team leads. Redundancy and Diversity was employed in the use of cross-discipline development teams as well as the use of part time subject matter experts. When required, a member of one team could be assigned to another team to supplement a critical capability in software development, for instance. Subject matter experts could also be brought in to provide critical capabilities that were lacking on a team at any particular point in time, and this capability was facilitated specifically to avoid long term use of expensive subject matter experts beyond their critical need. Elastic capacity was not employed to any meaningful extent as no issues existed that needed this capability: development teams were purposefully kept small so they could quickly react to changes, and only two spacecraft were required.

Discussion

Up to this point this article has principally addressed the practitioner. But there is work yet to do in research, and a few words might provide direction for some of what is still needed.

Part 1 and Part 2 of this article provide *some* framework and foundational considerations for

developing an agile systems-engineering life cycle model. Synergistic guidance from the work of others needs to be considered as well, particularly in the opportunity to address how agile systems engineering concepts might help in contracted development at fixed price and specification. Three bodies of thought emerging in the eighties offer some key perspectives that may have had more influence on what has happened since in various agility perspectives than is explicitly recognized, and in any event merit re-evaluation against today's understandings and objectives.

The first perspective came in 1981 with the publishing of *Systems Thinking, Systems Practice*, (Checkland 1981), questioning the application of rigid systems engineering practices to a class of systems that don't appear amenable to logical thinking, yet they are pervasive in the systems around us that have multiple stakeholders in various evolving states of satisficing for the moment. Checkland went beyond questioning, offering an alternative approach now known as soft systems methodology.

The second (chronologically) perspective came in a 1986 Harvard Business Review paper, *The New New Product Development Game* (Takeuchi and Nonaka 1986), acknowledged in (Sutherland and Schwaber 2007: 6) as sparking the thoughts that led to Scrum. That paper profiled a general process that engineered breakthrough innovative products composed more of hardware than of software, and exposed the role of what they called "subtle management", which affected product outcome by working behind the scenes to constantly rebalance diversity within the development teams. This concept is ignored by Scrum, yet crucial to the success of a rapid agile learning process.

The third perspective came in 1988 with the publication of *A Spiral Model for Software Development and Enhancement* (Boehm 1988). This marked a new turn of thought, offering an iterative, incremental alternative to the sequential waterfall approach, subsequently refined in a fundamental view (Boehm 2000).

Oversimplifying, Checkland put a focus on people, Takeuchi and Nanoka put a focus on product, and Boehm put a focus on process. All were concerned with uncertain and unpredictable engineering efforts. Each of these developments in the '80s gave legitimacy to, and spurred interest in, questioning the old ways and exploring new paths; paths meant to deal with uncertain and unpredictable operational environments.

The '90s might be viewed as a period of gestation, experimentation, research, and discovery.

It is suggested here that at the turn of the millennium three more bodies of synergistic relevant thought emerged. The first perspective came early in 2001 with the publishing of "*Response Ability – the Language, Structure, and Culture of the Agile Enterprise* (Dove 2001); which organized the agile systems research findings of the '90s into domain-independent enabling fundamentals for agility in engineered systems of any kind. The second perspective came later that same year with the publication of *Agile Software Development with Scrum* (Sutherland and Beedle); detailing a systems engineering management process for agile software development, and reviewed in this article for its agile enabling core. The third perspective came in 2002 published as *Agile Software Development Ecosystems* (Highsmith 2002); notable for its sane and revealing coverage of the principle software development practices sharing the agile family name at that time.

The six references suggested were ordered by their first appearance, and included because of the line of thinking they initiated. There is no pretention that they encompass all of the thought that needs consideration for developing an agile systems-engineering life cycle model, nor suggestion that seminal new thinking won't continue to emerge.

It is time to develop an agile systems-engineering life cycle model. This model, if a single one is sufficient, must take into account at least three different types of systems engineering, articulated well in (Sheard 2000): Discovery (very high complexity in problem space), Programmatic (complexity in solution space and possibly organizational), and Approach (complexity in the variation of applications, and possibly product lines).

An agile systems-engineering life cycle model might start with the framework displayed in Figure 2, and be guided by (ISO/IEC TR 24748-1 2010) toward identifying fundamental principle-based activities and processes that provide agility, independently as well as collectively, across all stages that warrant an agile approach. This model might justify the application of these principles, activities, and processes by identifying common systems-engineering environmental situations in need of agile response capability. Ideally, the model would be supported with case studies in a variety of systems engineering domains.

Conclusion

Unique to this article is the suggestion of a parallel between Peter Checkland's Soft Systems Methodology and the situation faced when an agile systems-engineering process is appropriate; the introduction of a ISO/IEC 15288 compatible life cycle framework for agile systems-engineering; an examination of the source of agility in the popular process known as Scrum; an examination of JHU/APL's CubeSat agile systems-engineering process for developing hardware/software systems, albeit an engineering project with more latitude than fixed price and requirements projects; and finally; a foundation for next steps toward developing a timely domain-independent agile systems-engineering life cycle model.

The growing acceptance and adaptation of agile software development methods has passed the tipping point in the software world, and is now motivating expectations in broader domain-independent systems engineering. The popularity of Scrum as a project management process, and the Siren song of the Manifesto for Agile Software Development, has created for some an expectation of a clear path to broader application. On the opposite extreme are those who make a clear case for inapplicability, e.g., (Carson 2012). Neither camp is actually focused on agility, but rather on specific software development management practices and principles that share *agility* as a family name.

There is no reason to expect domain specific software development practices to be applicable in domain independent systems engineering. For a simple disconnect example, (Carson 2012) observes that in software development the code designer is also the code fabricator; whereas in hardware, the designer's product is an intermediate document that is intended to drive the separate activities of a fabricator with a different world view. Integrated product teams attempt to address some of the communication issues, but inherently hardware design effort and fabrication effort are sequential activities of different time durations and different costs – at least currently in this very early period of automated 3D fabrication capability.

The ball is in motion toward the goal of an agile systems-engineering discipline. Perhaps many different balls are in motion, as the pressure to do systems engineering under accelerating environmental dynamics isn't waiting for a common disciplined understanding. We should, as practitioners and as researchers, identify and define design and operational guidance for adaptive system engineering processes.

Acknowledgements

The authors want to thank INCOSE Fellow Ron Carson in particular, as well as JHU/APL and unknown submission reviewers, for meaningful critical comment and improvement advice. Some of these suggestions could not be addressed appropriately within the constraints of this publication, but they warrant, and will guide, attention in subsequent opportunities.

References

Boehm, Barry. 1988. A Spiral Model for Software Development and Enhancement. IEEE Computer. May.

Bohem, Barry, 2000. Spiral Development: Experience, Principles, and Refinements. Special Report CMU/SEI -2000-SR-008 Edited by Wilfred J. Hansen (July), from workshop annotated slides presented at the Spiral Development Workshop, February.

Carson, Ronald. 2013. Can Systems Engineering be Agile? Development Lifecycles for Systems, Hardware, and Software. INCOSE International Symposium, Philadelphia, PA, 24-27 June.

Checkland, Peter B. 1981. *Systems Thinking, Systems Practice*. John Wiley, Chichester, UK.

Checkland, Peter and Sue Holwell. 2004, "Classic OR and "Soft" OR – an Asymmetric Complementarity. Chapter 3 in *Systems Modeling Theory and Practice*, Ed. Michael Pidd, Wiley.

Cockburn, Alistair. 2008. Using both incremental and iterative development. STSC CrossTalk (USAF Software Technology Support Center) 21, no. 5: 27-30. www.crosstalkonline.org/storage/issue-archives/2008/200805/200805-Cockburn.pdf

CubeSat. 2013. Revision 13 CubeSat Design Specification Provisional Release, August 19, 2013. www.cubesat.org/index.php/documents/developers .

Dove, Rick. 1998. Realsearch: A Framework for Knowledge Management and Continuing Education. In proceedings IEEE Aerospace Conference. Aspen, Colorado. 28 March. www.parshift.com/Files/PsiDocs/RealsearchIEEE.pdf

Dove. Rick. 2001. *Response Ability – The Language, Structure, and Culture of Agile Enterprise*. Wiley.

Dove, Rick. 2005. Fundamental Principles for Agile Systems Engineering. Conference on Systems Engineering Research (CSER), Stevens Institute of Technology, Hoboken, NJ, March. www.parshift.com/Files/PsiDocs/Rkd050324CserPaper.pdf.

Dove, Rick and Ralph LaBarge. 2014. Fundamentals of Agile Systems Engineering – Part 1. International Council on Systems Engineering IS14, Las Vegas, NV, 30-Jun-03Jul. www.parshift.com/s/140630IS14-AgileSystemsEngineering-Part1.pdf

Fowler, Martin and Jim Highsmith. 2001. The Agile Manifesto. Dr. Dobb's Journal, August. www.drdobbs.com/open-source/the-agile-manifesto/184414755.

Highsmith, Jim. 2002. *Agile Software Development Ecosystems*. Addison-Wesley Professional.

Huang, Philip M., Andrew A. Knuth, Robert O. Krueger, and Margaret A. Garrison-Darrin. 2012. Agile hardware and software systems engineering for critical military space applications. In SPIE Defense, Security, and Sensing, pp. 83850F-83850F. International Society for Optics and Photonics.

ISO/IEC 15288:2008(E), Systems and software engineering — System life cycle processes, Second edition 2008-02-01, Software & Systems Engineering Standards Committee of the IEEE Computer Society.

ISO/IEC TR 24748-1. 2010. Systems and software engineering — Life cycle management — Part 1: Guide for life cycle management. First Edition October 1.

Schwaber, Ken, and Mike Beedle. 2001. *Agile Software Development with Scrum*. Prentice Hall.

Schwaber, Ken and Jeff Sutherland. 2013. The Scrum Guide. www.scrum.org.

Sheard, Sarah A. 2000. Three Types of Systems Engineering Implementation. Symposium of the International Council on Systems Engineering, Minneapolis, MN, July.

Sutherland, Jeff, and Ken Schwaber. 2007. The Scrum Papers: Nuts, Bolts, and Origins of an Agile Process. Draft 10/14/2007. Scrum Foundation. http://scrumfoundation.com.

Takeuchi, Hirotaka, and Ikujiro Nonaka. 1986. The new new product development game. Harvard business review 64, no. 1: 137-146.

Biography

Rick Dove is CEO of Paradigm Shift International, specializing in agile systems research, engineering, and project management; and an adjunct professor at Stevens Institute of Technology teaching graduate courses in agile and self organizing systems. He chairs the INCOSE working groups for Agile Systems and Systems Engineering, and for Systems Security Engineering. He is author of *Response Ability, the Language, Structure, and Culture of the Agile Enterprise*.

Ralph LaBarge is a Principal Professional Staff member of The Johns Hopkins University Applied Physics Laboratory where his experience spans systems engineering, digital signal processing and cyber security. He received master's degrees in Computer Science, Electrical Engineering, and Information Assurance from The Johns Hopkins University, and a bachelor's degree in Electrical Engineering from the University of Delaware. He is currently enrolled in a doctoral program at George Washington University in systems engineering.