Synchronizing Systems Engineering and Implementation in Lean-Agile Programs

Robert Maione
Techna Systems Inc.
703-989-7977
rmaione@technasystemsinc.com

Mark A. Wilson
Strategy Bridge International, Inc.
(703) 297-4632
mwilson@strategybridge.com

Abstract. In traditional system development programs, high-level statements of capability and mission needs are elaborated and functionally decomposed, and then systematically allocated to discrete program segments or subsystems. Functional decomposition continues to be a classic systems engineering technique to define and manage the development of large systems, but this approach can create issues in systems that are being developed using Lean-Agile methods. Lean-Agile developers typically adopt a “time box” perspective, where the goal is incremental delivery of system capability on a predictable schedule. In other words, Lean-Agile developers may adopt a development rhythm with increments of rapidly evolving capability delivered every 10-12 weeks, while the SE team stays focused on the progress of the overall system as it was originally planned and defined. This divergence of both pace and perspective can lead to misunderstandings between systems engineers and developers or worse, disconnects between what the systems engineers think the developers are building and what developers are actually building.

Communication suffers when Lean-Agile developers and program systems engineers operate in separate organizational “silos” instead of working together as a cohesive team. Day-to-day implementation decisions with potential impact to overall system capability can become buried within lower level engineering and development documents that are hard to synthesize to support timely program decisions. In our work with large programs employing Lean-Agile development strategies, we discovered an approach that maintains systematic collaboration between developers and the SE team. In our approach, we explicitly measure and communicate incremental delivery of value using a new level of abstraction that we call Mission Value Threads (MVTs). MVTs directly link system requirements and their associated SE artifacts to Lean-Agile development team backlogs and design documentation. This paper describes how to use MVTs to manage value delivery while fostering better collaboration between developers and systems engineers.

Lean-Agile Development Context

Our Perspective. The authors of this paper are systems engineering practitioners. We cut our teeth on large, complex aerospace and defense programs, and we have subsequently spent decades working as consultants for both private industry and the U.S. federal government. We will neither identify nor reveal specific details about any program or project on which we have worked, since that would betray our clients’ confidence in our discretion. However, we believe that we have some lessons learned to share with the systems engineering community that are worthy of further discussion.

Our purpose in writing this paper is to describe a technique that we introduced on two Lean-Agile programs to manage value delivery and foster better collaboration between developers and systems engineers. We think this technique holds great promise, especially on large, complex, software-intensive programs where the eventual solution must integrate with other systems. We do
not suggest to the reader that a Lean-Agile development approach should be used on every program or project. We are enthusiastic advocates of strategic thinking who believe that every Program/Project Manager and Lead Systems Engineer should carefully decide on the solution delivery strategy, development strategy, and systems engineering implementation approach that offers the best opportunity to achieve a favorable outcome. Moreover, Lean-Agile methods are not appropriate for every situation. For example, if the solution “customer” is not willing to accept an Agile development philosophy with incremental deliveries of value based on a planned development rhythm, or if that customer is not willing to commit to active, continual participation throughout the project’s development, then forcing a Lean-Agile approach is not likely to succeed.

**Agile Development.** Agile development is not a new idea, and, contrary to what many people believe, its roots are not in software development. Hirotaka Takeuchi and Ikujiro Nonaka published a paper, *The New New Product Development Game*, in Harvard Business Review in 1986 that should be required reading for all systems engineers. In the paper, these two university professors outlined the development principles that drove innovation on the FX-3500 medium-sized copier (introduced by Fuji-Xerox in 1978), the PC-10 personal-use copier (Canon, 1982), City Car with 1200 cc engine (Honda, 1981), the PC 8000 personal computer (NEC, 1979), the AE-1 single-lens reflex camera (Canon, 1976), and Auto Boy, known as the Sure Shot in the United States, lens shutter camera, (Canon, 1979). Takeuchi and Nonaka wrote about the essential differences between traditional, sequential development and a different (“new”) approach. Many of the same principles described by Takeuchi and Nonaka in 1986 would later be elaborated by the Agile Alliance in their 2001 *Agile Manifesto*, which was focused on software development. Jeff Sutherland and Ken Schwaber, the co-creators of Scrum, a popular Agile methodology used in many software projects, dedicated their *Scrum Papers* to Takeuchi and Nonaka, crediting them for helping to transform software development worldwide.

Agile methods rely on competent development teams who work closely and collaboratively with their customer(s). These Agile development teams rely on short, incremental build cycles (two to six week “time-boxes”), continual delivery (flow) of value to users, and continual communication with all key stakeholders. In contrast, traditional, sequential development methods (e.g., Waterfall, Vee) tend to rely on rigorous up-front planning, comprehensive design guidance (requirements) before implementation begins, scripted communication driven by formal reviews and documents, and strict baseline change control.

**Lean Principles.** Just as Agile Development didn’t suddenly appear in 2001, Lean Manufacturing principles were being implemented and refined long before James P. Womack and Daniel T. Jones documented Lean principles in their book, *The Machine That Changed the World* in 1990. Perhaps Henry Ford (Ford Motor Company’s assembly line) and Kiichiro Toyoda (the Toyota Production System) deserve much of the credit for modern Lean thinking, but Womack and Jones successfully distilled this philosophy into five Lean principles:

- **Specify the value desired by the customer**
- **Identify the value stream for each product providing that value and challenge all of the wasted steps (generally nine out of ten) currently necessary to provide it**
- **Make the product flow continuously through the remaining value-added steps**
- **Introduce pull between all steps where continuous flow is possible**
- **Manage toward perfection so that the number of steps and the amount of time and information needed to serve the customer continually falls**

(Lean Enterprise Institute, [https://www.lean.org/WhatsLean/History.cfm](https://www.lean.org/WhatsLean/History.cfm))

**Experience Base.** We have observed—first-hand—programs over the past decade that have attempted to use Agile development methods, but struggled to deliver acceptable solutions within cost and schedule constraints. In every case, these so-called Agile programs started off with smiling customers and optimistic developers, but as they moved through their life cycles and risks
continued to manifest, many of these programs experienced significant cost overruns and schedule delays—which was the point when we typically got involved. As we reflect on our past experiences, we suggest there are two basic ways to implement an Agile development approach that will lead to unhappy stakeholders: Unconstrained Agile and WAGILE (Figure 1). An “Unconstrained Agile” development approach often fosters continual requirements creep. Many “Unconstrained Agile” programs cannot adequately demonstrate that the value delivered justifies the resources being consumed, and that situation usually results in a program restructure or cancellation. At the other extreme, we’ve seen programs try to combine traditional sequential (i.e., Waterfall) systems engineering processes with Agile development processes, using an approach some have termed “WAGILE”. In many WAGILE programs, the systems engineers are frustrated by a perceived lack of process discipline exhibited by the developers, and the developers feel suffocated by their perception of a slow, cumbersome change management process. When these frustrations reach a tipping point, communication breaks down and program failure is not far behind. On several WAGILE programs, we witnessed development teams become so frustrated with the plodding pace of program governance, particularly engineering and configuration management reviews and decision boards, that the developers’ goal became to avoid interactions with the systems engineering teams altogether. Waiting for governance decisions slowed development to the point where acquisition project managers decided to ignore systems engineering concerns and instead, they relied on direct input from end users on new features with no overarching, system-level guidance. The result was quarterly software deliveries presented to program control boards whose members had limited knowledge of the capabilities included in those releases. Frustrated program executives found themselves approving new software releases with content dominated by end user demands, with little to no strategic content deemed necessary by leadership to address long-term, enterprise goals.

Figure 1. Models for Implementing System Engineering in Agile Development Projects

On two large, software-intensive programs where we had the ability to influence the development approach early in the acquisition process, we convinced our customers to adopt a more balanced and responsive systems engineering and Agile development approach. Our goal was to allow developers to use Agile principles to maximize development efficiency and effectiveness while
staying closely connected to system and system of systems architectures. We investigated how Lean principles combined with Agile development (Lean-Agile development) could enable improvements in both process efficiency and product quality. We call this a Hybrid Lean-Agile model (Figure 1), where development teams effectively collaborate with systems engineers to elaborate requirements and make timely decisions synchronized with Lean-Agile development team cadence.

Figure 2. Comparison of Two Programs Using Hybrid Lean-Agile Development

Figure 2 illustrates the Hybrid Lean-Agile development approach on those two, software-intensive programs. In the first program, the Lean-Agile approach helped the team to prioritize over 1400 “features” that software developers derived from allocated component requirements (Figure 2, System A – boxes 2A and 3A). In this program, the decision to use an Agile methodology was made during the process of defining system requirements. During this time, the entire program support team was trained on the Scaled Agile Framework (SAFe) Version 3.0. (Authors’ note: this was several years prior to the incorporation of Lean principles in SAFe 4.0.)

Given our limited practical experience with Agile and SAFe at the time, the program support team decided to build a program backlog by having the software development teams derive features from the allocated component requirements. In retrospect, this decision was a mistake that we realized only after the development teams completed their task and we saw that we had too many features. We anticipated the requirements baseline would evolve with the design, and our intent was to establish a program baseline before development started so we could measure program performance. However, allowing each of the software teams to independently derive features without any idea of how those features were linked resulted in a wide variation of feature details and too many features for the customer to prioritize at increment planning meetings. To address this issue, we advised our customer to develop a vision for what they wanted to see after the first six
months (two program increments). The vision was then used to sort the features and reduce them to a manageable set (see Figure 2, System A – boxes 4A and 5A). This “vision statement” was the genesis for our Mission Value Thread (MVT) concept.

Building on the experiences and lessons we learned from System A, we designed the development approach for a second program (System B) that incorporated an Agile methodology from the start. To address the issue of too many unconnected features, we developed the requirements and MVTs together, early in the system definition phase (Figure 2, System B - box 2B). The linkages we established allowed the systems engineering team to produce requirements documents (a contract deliverable), while at the same time, systems engineers and software developers worked together to craft a plan for implementing the requirements with multiple, end-to-end value deliveries using time-boxed increments (Figure 2, System B - box 3B). In addition, we built into the program plan a process that maintained systems engineering and software development collaboration and kept systems engineering and software development processes in lock step.

**Mission Value Threads**

**Definitions.** An MVT is a discrete capability (product or service) that customers (or users) find inherently useful that can be delivered within a single, Agile Program Increment (PI), typically lasting 10-12 weeks. In other words, an MVT is a relatively small increment of value, often cutting across system components (C1, C2…Cn), with well-defined acceptance criteria (Figure 3). Each MVT defines an integrated, fully verified, end-to-end system capability. By contrast, a traditional, linear development (e.g., Waterfall) approach relies on a system requirements document (SRD) that is further decomposed to component requirements (CRD) that specify the system or component capabilities to be delivered and verified for overall system acceptance that often occurs years after program start. Since Lean-Agile development is built around regular deliveries of incremental capability every 10-12 weeks, MVTs enable us to measure discrete slices of functionality from the planned overall system capability.

Our concept of an MVT is different from what many agile practitioners and SAFe 4.0 define as an “Epic.” Epics are often broadly defined as a vision for a large effort of work, or what can be generally thought of as a “big user story.” In the *SAFe Reference Guide*, Epics are defined as:

> …containers for significant initiatives that help guide value streams toward the larger aim of the portfolio…by their very nature, portfolio Epics are large and tend to be abstract, so defining success criteria is a capstone to establishing a shared understanding among stakeholders of what the Epic really implies for the business.

The *SAFe Reference Guide* also states that …it is often not necessary to complete Epics….and it is sometimes the case that implementing some – but not all – of an Epic achieves the bulk of economic value.

In our world where outcome-focused contracting is the norm, one cannot contractually obligate a developer to deliver an Epic that is defined as an “abstract statement of desired value.” (The authors have also observed Epics used on level-of-effort (LOE) development contracts, but those programs generally end up as Unconstrained Agile programs as discussed earlier in this paper.) In contrast to an Epic, an MVT is a verifiable thread of working system functionality that is delivered at the end of a PI. MVTs that are on the program backlog will be completed, and the MVTs must have a “Definition of Done” that drives verification and validation during integration and verification activities. In other words, MVTs set an expectation of value that can be measured and assessed upon delivery.

MVTs provide us a tool to synchronize and align Lean-Agile implementation activities with systems engineering activities to provide insight and visibility that enables us to assess the health of
A Lean-Agile development strategy facilitates in-process validation through rapid, incremental deliveries of discrete, end-to-end capability to users. The users, in turn, are obliged to provide timely feedback to developers who are continually shaping the system to meet evolving needs. This dynamic development environment cannot slow implementation progress simply to give the systems engineering team time to update their SE artifacts. Moreover, SE artifacts such as the System Requirements Document or the Component Requirements Documents (SRD and CRD in Figure 3) do not define requirements around partial deliveries of system capability. A major advantage of using MVTs is that they are easy to understand across all levels of the program and can be rapidly evaluated and prioritized to make value-based decisions.

**Figure 3: Mission Value Threads (MVTs)**

**MVTs and Partial Requirements.** System requirements describe an end state – what the system must be or do at completion. In contrast, MVTs are discrete capabilities – increments of user value that can be traced to component and system requirements – that usually emerge from the interactions of several subsystems or components at a particular time phase of development. In Lean-Agile development, the goal is to incrementally deliver capabilities based on time-boxed efforts that partially satisfy requirements. Through techniques such as activity diagrams and use cases, we can elaborate MVTs by defining the actions required and mapping the activities to the architecture components needed to realize the end-to-end value described by a given MVT. As MVTs are defined and elaborated, they can be linked to system requirements (e.g. functional, performance, security, or other non-functional requirements). The MVTs thus provide the ability to link partial requirements satisfaction to incremental deliveries. They provide the SE team insight on exactly what the development organizations are building. As more MVTs are delivered, adjustments or refinements to system requirements can be based on an actual working system at a given point in the lifecycle.

**A Visual Analogy.** A way to visualize the MVT construct is using a multi-color layer-cake analogy shown in Figure 4. The cake represents the system with its total planned capability. Each uniquely colored layer represents one element or component of the system; the frosting between layers represents the interfaces between components. In traditional SE terms, the cake represents the system product breakdown structure (PBS), with each cake layer representing one segment or subsystem of the PBS. Collectively, the layers of the cake will provide all planned capability, but a wedge (slice) of cake represents a discrete increment of capability (MVT) that can be delivered in one PI.

Systems engineering and hardware and software implementation teams, in concert with end users, elaborate and execute MVTs (represented as slices of cake). Each MVT is implemented as a fully
integrated and tested “slice” of capability with end-to-end value realized and demonstrated prior to beginning work on the next Program Increment. Contrast this approach to traditional, sequential development, where requirements are established early in the program lifecycle, put under baseline control, and then each of the system elements (cake layers) are “baked,” often independently by different organizations. Traditionally, the integration of the completed system elements (cake layers) occurs after implementation and is verified at major Integration and Verification milestones, usually many months after the start of the development effort. These integration and verification activities are also when we typically encounter major technical problems. Pacing system development by using MVTs avoids many of these nasty integration surprises because MVTs are fully integrated and verified increments of capability delivered every 10-12 weeks. Thus, interface issues can be discovered much earlier in the development lifecycle, and each PI delivers functionality that can be validated with the program office and end users. This incremental approach reduces program risk because issues are addressed early, when changes are typically easier and less expensive to make.

Figure 4: System Capability Represented as a Layer Cake

Requirements in Lean-Agile Programs

Programs often rely on highly structured requirements documents to communicate desired value. However, the overall context from which the requirements were derived (e.g., CONOPS) may not be understood by development teams or may be difficult to stitch together from other reference documents. Recall that a key attribute of Agile development is fostering face-to-face communication with end users. However, allowing users to drive development activities without ensuring that the systems engineering team keeps the “system perspective” foremost in everyone’s mind is not a recipe for a successful system solution. While we believe that system requirements (at some level) are necessary for all programs (even Agile programs!), we argue that they are not sufficient to adequately create plausible Lean-Agile implementation plans. Managing to MVTs can help the program team translate high-level requirements and context into detailed implementation plans.

A key challenge in any Lean-Agile system development is maintaining systematic, bidirectional communication between the SE team and developers. The SE team may have defined a high-level system scope, a CONOPS, or interfaces before development began, but the developers are continually evolving the system requirements and design as a consequence of their Agile approach to system implementation. It is imperative that the SE team’s perspective stay connected to the evolving system, especially design decisions that impact the system architecture, or we risk over-emphasizing a narrow user perspective in the final system. Worse, we risk losing the
perspective that links the system under development to the higher-level system of which it will eventually become part. Stated simply, we don’t want the developers and users to cut off the SE team’s input during development.

Development teams often have insufficient insight into planned end-to-end system capability and less insight to the relative priority of discrete capability increments desired by the enterprise or its end users. Consequently, developers are often left to derive value and delivery priority on their own. While current software management tools like JIRA could potentially be programmed to include interactions with systems engineers, we haven’t seen explicit links or capabilities that are designed with the idea of synchronizing systems engineering and software development within a typical Agile PI cycle. We believe that software engineers who are writing stories in preparation for the next Sprint or PI should be working with (and elaborating) artifacts that are collaboratively developed with the SE team prior to the start of Sprint or PI planning. However, based on our recent investigations, interactive interfaces between popular SE and software development management tools do not yet exist.

**Integrating SE and Developer Tool Sets.** During the requirements definition phase of a typical development program, multiple levels of requirements are elicited, engineered, and documented to align the system of interest to the strategic vision of the customer, stakeholders, mission, and end user goals and needs. We have worked on Lean-Agile programs where lower level software requirements (typically captured as User Stories in development management tools such as JIRA) are never traced to system requirements. It can sometimes seem as though the systems engineers and developers live in separate, parallel but unconnected universes. The respective tool suites used by systems engineers and developers don’t appear to communicate well, either. For example, systems engineers may use DOORS to manage requirements, but the developers use tools like JIRA or Rally to manage their daily work, and those tools typically aren’t connected to DOORS.

In the absence of automated tools, communication is dependent on face-to-face meetings and manual exchange of documentation. On the programs where we’ve used MVTs, SE artifacts were generated using SE tools and hyperlinks to those artifacts were established in software development management tools. This approach was cumbersome when artifacts were modified or further elaborated during design evolution, requiring a labor-intensive manual process to keep the tools synchronized. Seamless bidirectional communication can only be achieved once SE tool suites (e.g., DOORS, CORE, GENESYS, MagicDraw) are integrated with the project management tool suites that the developers use to manage their day-to-day work (e.g., JIRA, Rally). The SE tools used today are, at best, loosely coupled with developer tools, typically through hyperlinks to static images of models. This slows the flow of information between systems engineers and developers, and that is a formula for a communication breakdown. For example, a systems engineer may create a high level or initial model of a desired system function, but a software engineer may modify the model as the software is developed and demonstrated to users at Sprint and PI demonstrations. Absent an automated way of communicating between SE and development, the “as built” system may diverge from its original vision. Given automated tool linkages, the builder and customer organizations could have automated access to detailed development status as well as design changes, updated daily by Lean-Agile developers in their workflow management software. This would also provide precise information on overall value creation measured by MVTs.

**Time Box Abstractions**

**Decomposing Work: Solutions, Initiatives, Features, Stories, and Tasks.** Once an initial MVT elaboration has been created to fully describe all linked actions, systems engineering teams should collaborate with development teams to ensure that the MVT elaboration is properly captured and that the teams agree on allocated requirements and actions. This is a perfect opportunity to collaborate on MVT “sizing.” Earlier in this paper we defined an MVT as a discrete capability that
customers (or users) find inherently useful that can be delivered within a single PI. Who else beside the developers can assess whether an MVT can be completed within a single PI, since they are the ones who will have to do the work? However, project constraints may require that MVTs be broken up into smaller pieces to fit them into a PI due to other competing priorities. We do not recommend decomposing the MVT into “smaller” and separate MVT abstractions because that obscures the fundamental concept of a MVT as an increment of system capability. Rather, we have introduced a second level of abstraction that we refer to as a Solution (see Figure 5). A Solution is a portion of a MVT that can be implemented in one PI. Like an MVT, a Solution is fully integrated and tested in one PI, but it gives program managers and development teams the flexibility to right-size work to maintain continuous flow (a Lean principle), while limiting the number of MVTs that must be tracked. The Solution abstraction is only needed when reasonable MVT size estimates are not possible.

![Figure 5. Requirements Aligned to Needs](image)

Once the MVTs have been defined, SE and development teams collaborate to define more discrete levels of abstraction that we call Initiatives. Initiatives are portions of MVTs that are allocated to a specific component in the system architecture. Here we assume that system components are allocated to Agile Release Trains (ARTs); and each ART is assigned one or more components to develop. (An ART is an organizational construct, typically comprised of 50-125 people, that plans and executes work together. Reference: [www.scaledagileframework.com/agile-release-train](http://www.scaledagileframework.com/agile-release-train).) Initiatives are portions of an MVT assigned to an ART (or teams within an ART). The Initiatives become the tie points that help the SE and development teams align development and delivery of system capabilities expressed by their parent MVTs.

The development teams create and implement Initiatives, Features and Stories (illustrated in Figure 5) needed to realize MVTs. An Initiative is a bundle of features that can be implemented in one PI and implements a portion of a MVT. A Feature is a piece of an Initiative that can be implemented in one PI, but may require several iterations (e.g. Sprints lasting two to four weeks) to fully complete.
A Story is a portion of a feature that can be implemented in a single Sprint. A Task is a piece of a Story that can be completed in one or two days. All of these time-box abstractions serve to decompose and manage the development “work” to implement both planned and continually evolving system capability. In contrast, a traditional, top-down, sequential development approach (e.g., Waterfall) focuses on pre-planned capability assigned to pre-defined system elements (“products”), all of which is under strict change control. This is a key difference between traditional development approaches that are focused on delivering fully integrated “products” that meet requirements, while a Lean-Agile approach focuses on incrementally realizing MVTs (representing increments of capability) using fixed development time boxes.

Implementing MVTs on a Development Program

Architecture and Design. The terms “architecture” and “design” are sometimes confused. When we refer to architecture in this paper, we are referring to the high-level instantiation of the solution concept. Architecture decisions address strategy, purpose, and structure: how system elements and components will interact with each other and other systems. Sound architectures evolve as a consequence of decisions that address a wide range of factors, from long-term strategic goals to specific implementation concerns that can impact the realization of those goals. When we refer to design in this paper, we are referring to a specific solution implementation. For example, the systems engineering team on a large software project may make an architectural decision about whether to use a given technology, while the developers will make specific design decisions about how to implement data structures or algorithms consistent with that technology.

Architecture and design concerns often overlap. For example, a software engineer’s design of a virtual private cloud configuration might impact the application’s ability to share data with external systems that have not yet been defined, but are being planned by systems engineers for future acquisition and development. In the early stages of many programs, systems engineers make architectural decisions with little input from the implementation teams who will actually build the hardware or write the software to realize the system. In our view, this sequential (stovepipe) model of architecture definition and implementation must become more collaborative, particularly for software-intensive systems where technology and user needs change rapidly. For example, one major software program on which we consulted switched mid-stream from a customer-provisioned hardware implementation to a cloud-based implementation. On that program, the systems engineering team required significant input from software engineers who had practical experience with cloud storage and metadata services, to redefine the solution architecture while satisfying stressing throughput and timeliness requirements.

The CART. In traditional program offices, systems engineering, integration and test, developers, and program management staff are often organized into functional silos. This organizational construct (“cylinders of excellence”) requires defined interfaces among functions that may inhibit timely and effective communication. Instead, we recommend organizing the program staff into a cross-functional, Lean-Agile team that we call Collaborate Across Release Trains or “CART” (see Figure 5). The traditional Agile Release Train (ART) construct is focused on development activities. The CART, however, includes systems engineering, program management and development representatives to increase the efficiency and effectiveness of the overall development and system integration process. In a Lean-Agile development project, development work is paced by “backlogs” at the program, project and team levels. If something is on the program backlog, it will be built. Conversely, if a desired piece of value isn’t on the program backlog (or isn’t properly articulated in lower level backlogs), it is likely that the desired capability will either not be built, or will be built incorrectly. The CART facilitates team collaboration to ensure that the systems engineering artifacts and other program information remain connected to MVTs and flow from the program backlog to all lower level development backlogs. Stated another way, the CART ensures that the systems engineering perspective and function remain relevant to the program, despite the
evolving requirements and design baselines that are characteristic of every Lean-Agile development.

The key to the CART is the interaction and planned communication between systems engineering and development teams illustrated in Figure 6. The CART facilitates the synchronization of systems integration activities to incrementally deliver end-to-end value. The CART executes a streamlined, integrated, and repeatable Lean-Agile process by orchestrating monthly Iterative Design Sessions (IDSs) to elaborate MVT details and review design details in planning for an upcoming PI. The IDSs typically include reviews of activity diagrams, behavior models, and other design artifacts relevant to the next PI. The CART also reviews project-level backlogs to plan and prioritize future PIs. (Authors’ note: we define an ART Initiative and Feature backlog as the project backlog.) The project backlogs and implementation plan offers additional levels of insight, enabling the program office to see how feedback provided at working sessions (that should occur between IDSs) is addressed by the development teams. The transparency provided through this approach maximizes team velocity and ensures high quality team deliveries.

![Figure 6. CART Communication Process](image)

Current Agile methodologies, such as the Scaled Agile Framework (SAFe), suggest planning for the next PI at the end of the current PI in an Innovation and Planning (IP) Sprint. We argue that a two-week IP Sprint at the end of a PI is insufficient to properly engineer programs of even moderate complexity. Our idea for monthly CART IDSs gives the program management and SE teams ample opportunity to provide timely feedback on development and transition activities. The IDSs foster more thorough implementation planning, with an emphasis on reaching agreement on dependencies across ARTs needed to achieve end-to-end value. The outcome is accelerated systems engineering and development team velocity. With IDSs preceding IP Sprints, more time is spent on creating good designs, which increases the effectiveness of PI planning and leads to higher quality developments.

**CART Structure and Cadence.** The structure of the CART depends on the needs of each individual program, but CART membership must, at a minimum, include Systems Engineering, Integration and Test, and Program Management staff along with development leads such as Product Owners and lead Software Engineers. In general, we recommend that one CART subgroup focus on future PIs by elaborating MVTs and fostering collaboration across ARTs. Another CART subgroup should focus on the delivery of the current PI, to include monitoring MVTs and test activities to verify requirements. Additionally, a third CART subgroup should focus on the evaluation of the just delivered PI, validating MVTs with users and assessing functionality and performance prior to
transition into operations. We execute the CART with Lean-Agile methods such as Kanban or Scrum, with cadence and deliverables aligned to support PI planning and development deliveries.

**CART Benefits.** MVTs enable agile development teams to right-size work and measure progress through the continuous delivery of working software and hardware while staying connected to system requirements. The CART provides a systematic process to align systems engineering activities to the Lean-Agile development process using MVTs as the common tie points. CART subgroups collaborate in the development of detailed designs in synchrony with the Agile development cadence. Between IDSs, systems engineers can focus on creating detailed systems engineering artifacts (e.g., sequence and state diagrams) to define architecture details. As the system architecture evolves as a consequence of the development activities (a key principle of Agile development), the system models can be easily updated to reflect the as-built architecture. The value of these SE artifacts is that they are tied directly to the development efforts to proactively communicate solution intent as development proceeds. These artifacts also provide important documentation necessary to sustain system capabilities after solution acceptance and eventual transition into operations. Additionally, the CART performs the control functions traditionally managed by separate Configuration Control and Engineering Review Boards. This approach streamlines change control as part of CART day-to-day responsibilities, better matching the rhythm and pace of Lean-Agile development teams. By including key program office stakeholders in the CART, changes to the architecture and requirements can be made at the same pace of Lean-Agile development teams, and all stakeholders are fully aware of the intended value for PI deliveries streamlining the approval of new capability into operations.

**Summary**

Complex system solutions must be engineered and not merely be allowed to emerge as a consequence of implementation. A Lean-Agile development strategy can work for complex systems, but only if we ensure that systems engineering stays connected to – and keeps pace with – Lean-Agile implementation activities. Mission Value Threads (MVTs) can help us manage solution technical progress by providing a mechanism to measure incremental deliveries of value. MVTs thus provide programs a tool to synchronize and align Lean-Agile implementation activities with associated systems engineering activities. Then, by organizing program staff into a cross-functional CART team, we can better integrate systems engineering with solution implementation to increase both efficiency and effectiveness of the overall development process. The CART approach provides a process to elaborate system architecture and design details to ensure that solution architecture decisions are completely visible to all key stakeholders. Our experience suggests that MVTs and the CART organization construct can improve communication and avoid misunderstandings between systems engineers and developers, thus helping programs proactively manage incremental value delivery while fostering better program team collaboration.

**References**


**Biography**

Robert Maione is the President of Techna Systems Inc., a company that specializes in Lean-Agile best practices. He is certified as a Scaled Agile Framework (SAFe) Program Consultant by the Scaled Agile Academy and earned a B.S. in Computer Science from the New Jersey Institute of Technology.

Mark A. Wilson is CEO of Strategy Bridge International, Inc., a company that helps clients to develop strategies and plans that translate into bottom-line results. He is certified as an Expert Systems Engineering Professional (ESEP - INCOSE), Certified Manager of Quality/Organizational Excellence (American Society for Quality) and earned a B.S. from the U.S. Naval Academy and M.S. from the U.S. Naval Postgraduate School.