2017

Program Requirements: Complexity, Myths, Radical Change, and Lean Enablers

Bohdan W. Oppenheim
Loyola Marymount University, boppenheim@lmu.edu

Project Management Institute

WHITE PAPER

Program Requirements:

Complexity, Myths, Radical Change, and Lean Enablers

By Bohdan W. Oppenheim, PhD

Abstract

This paper presents a comprehensive discussion on the difficulties of formulating good and stable requirements early in complex engineering programs and the consequences on program execution. Formal classical systems engineering and program management (CSEPM) methodology is based on the assumption that the knowledge to anticipate all interfaces and create good requirements exists early in the program, and that it is a matter of working out the details to build extremely complex devices such as satellites, aircraft, refineries, nuclear power plants, and high-speed rail. The author argues that classical systems engineering and program management works well only for well-understood systems, but it breaks down when the knowledge of what needs to be done still needs to be discovered, which is the case with most complex systems. Once the requirements and interfaces are defined, flowed down, and allocated to all applicable lower levels, the primary mechanism for mission assurance is requirements verification—by test, analysis, inspection, or demonstration. In real programs that develop new systems, the reality leads to the following *Faustian* choice: Either develop and anticipate all interfaces and requirements early, when the knowledge is not yet available, hoping to make the systems engineering Vee process in one pass and to feed the requirements to numerous suppliers early; or conduct massive, painful, and cost- and schedule-busting requirements changes throughout the program, renegotiating the requirements with the suppliers. This paper traces various historical phases that contribute to the present difficulties with requirements; the author argues that in order to radically change this major deficiency of classical systems engineering and program management, a radical change of the business model is needed, such as that practiced by SpaceX. The paper proposes development of special Lean enablers (LEs) to be formulated for those readers who might be interested in this radical approach.

This radical change, though, is just not practical for large engineering programs that have to operate under complex political and financial constraints. T*he Guide to Lean Enablers for Managing Engineering Programs* (Oehmen, 2012) applies to these programs directly. The Lean enablers contain a well-integrated set of systems engineering and program management practices that have the potential to vastly improve the requirements development and related program execution. This paper discusses the applicable Lean enablers.

Table of Contents

1. Introduction

In 2012, after a two-year research and ground-breaking cooperation, Project Management Institute (PMI), International Council on Systems Engineering (INCOSE), and Massachusetts Institute of Technology (MIT) jointly published *The Guide to Lean Enablers for Managing Engineering Programs* (Oehmen, 2012). It represents the collective wisdom and consensus of 28 experts and about 180 practitioners. The 326 Lean-inspired best practices, called Lean enablers (LEs), were organized into 46 thematical groups under six Lean principles. Together, they represent the multi-disciplinary integration of program management, systems engineering, and Lean.[1] The team used surveys, analyses, and intensive interactions with experts of past programs as the main research instruments in the capture, development, and validation of Lean enablers.

This project was initiated in order to increase the level of validation of selected Lean enablers. This study is focused on requirements because they play a critical role in program formulation, execution, and value/benefit delivery to customer stakeholders in modern programs. It can be said that modern programs are driven by requirements. Yet experience from complex programs such as satellite, spacecraft, ship, nuclear power plant, high-speed rail, city infrastructure, healthcare, K-12 education, and many others demonstrates that formulation of good and stable requirements is a formidable task and is rarely successful. In 2011, the Government Accountability Office (GAO, 2011) published an astonishing statistic that, on average, 82% of requirements in recent defense programs are changed over the program lifecycle, which means that in spite of the huge effort, only 18% of the requirements released at program initiation remain stable—a rather devastating number. This statistic is one reason for notorious frustrations with large weapons and infrastructure programs, including those that exceeded program cost and schedule, required Nunn-McCurdy reviews,[2] and were terminated prematurely. Clearly, imperfect requirements are not the sole source of program troubles. Oehmen (2012) lists the following 10 major challenges in managing programs and each of them is capable of robbing a program of technical and/or business success:

1. Reactive program execution
2. Lack of stability, clarity, and completeness of requirements
3. Insufficient alignment and coordination of the extended enterprise
4. Value stream not optimized throughout the entire enterprise
5. Unclear roles, responsibilities, and accountability
6. Insufficient team skills, unproductive behavior, and culture
7. Insufficient program planning
8. Improper metrics, metric systems, and key performance indicators
9. Lack of proactive management of program uncertainties and risks
10. Poor program acquisition and contracting practices

Table 1 lists critical performance characteristics of nine recent and major U.S. Government space programs. The table data are based entirely on numerous GAO reports studied by Pabst (2014). The table lists unstable requirements as a major contributor to imperfect program performance in seven of the nine programs listed. In addition to unstable requirements, GAO (2014) lists the following other major reasons for program problems:

- Unstable program funding (which is usually the result of other problems in a given program);
- Starting the program before technology is sufficiently mature[3,4]; and

- Excessive complexity (so called "gold plating" of programs[5] by Ashton Carter, then Secretary of Defense for Acquisition and Logistics).

It is always desirable to correlate individual causes to program measures of success. Regretfully, the data quoted in **Table 1** represent too small a sample size to allow that. The principal investigator (PI) was informed by his high-level contacts that defense programs lack meaningful metrics of this kind. For example, government programs do not track requirements instability over a program lifecycle—a critical measure of program quality[6]; for these reasons, the scope of this paper is limited to program requirements.

PROG-RAM	Contracting Agency	Requirements Stable?	Funding Stable?	# of TRL <<6	Final Cost B$	Cost Growth %	Schedule Growth %	# of Nunn-McCurdy Reviews	Excessive Complexity?
SBIRS	Air Force	Unstable	Unstable	3	18.8	300%	120% Terminated	4	Yes
GPS IIF	Air Force	Unstable	Unstable	0	2.6	257%	133%	1+	No
GPS III	Air Force	Stable	Stable	0	4.2	2%	40%	0	No
GPS OCX	Air Force	Unstable	Unstable	14	3.695	28%	50%	N/A	Yes
MUOS	Navy	Stable	Stable	1	7.3	6%	20%	0	No
JMS	Air Force	Unstable	N/A	N/A	N/A	N/A	50%	N/A	Yes
SBSS	Air Force	Unstable	Stable	5	0.922	178%	60%	0	No
AEHF	Air Force	Unstable	Stable	11	14.372	154%	150%	3	No
NPOESS	Air Force, NOAA, NASA	Unstable	Stable	13	13.162	122%	Terminated	2	Yes

Table 1: Performance of selected major U.S. space programs (Pabst, 2014).

The classical systems engineering and program management approach has evolved significantly over the last 60 years, driven by two powerful forces: the uncompromising need for reliable system-level (including all sub-systems) performance and the inefficient government system acquisition practices and incentives. The evolution achieved high levels of program success (e.g., 60 successful space launches in the U.S. Air Force (Oppenheim, 2011)), but this technical success came at the expense of serious unintended consequences:

- Notoriously costly and long development programs that extend for years and even decades, not infrequently with reduced performance;
- Slow deterioration of programs from intended "great engineering and management" to unintended "bureaucracy of artifacts"; and
- The growth of several important myths regarding program management and systems engineering.

2. Unintended Emerging Properties of Classical Systems Engineering and Program Management

The systems engineering discipline was created by Si Ramo[7] and Dean Woldridge in 1954 during the ballistic missile program to help with the development and integration of complex systems that must work unconditionally (Jacobson, 2001). NASA's definition of systems engineering is "a methodical, disciplined approach for the design, realization, technical management, operations, and retirement of a system" (NASA, 2007, p. 3).

Systems engineering is supposed to be the practical engineering realization of *systems thinking*: a comprehensive and rigorous development process of the system that satisfies all stated needs during an entire system lifecycle. In complex systems, this includes many thousands of critical parts operating in hostile environments and created by several engineering and software disciplines. Usually, the biggest challenge is the integration of the parts into a functioning reliable system. Traditional, discipline-based engineering tends to focus on the development of parts and subsystems and does an imperfect job of integrating them at the multidisciplinary system level.[8]

As discussed in GAO (2014), recent programs have experienced highly significant problems related to unstable requirements, notoriously exceeding initial program costs and schedules, and often compromising the system performance or quantity of delivered systems.

2-1. Evolution of systems engineering toward detailed requirements

Johnson (2002) describes a fascinating evolution of aircraft procurement—from prototype-based to detailed requirements-based. Prior to World War II, the Army Air Corps (AAC) issued few detailed requirements. Generally, the detailed requirements were limited to only those defining the speed, altitude, maneuvering, payload, and endurance. Aircraft companies responded to the top-level requirements with a prototype aircraft designated as X. These were developed at the company's risk, usually on fixed-price contracts and evaluated by the Army Air Corps. The Army Air Corps would provide evaluations back to the company and the company would then build a Y prototype. The Army Air Corps would then evaluate the Y prototype before issuing a production contract. As World War II loomed, the U.S. government realized that the X and Y prototype systems could no longer produce the large number and variety of aircraft that would be required to fight the war. This was because of two factors. First, it took significant time to develop and evaluate the X and Y prototypes. Second, the risk of developing an aircraft under a fixed-cost contract was so large an undertaking to industry that the fledgling aircraft companies could not afford to have multiple aircraft under development simultaneously. As a result, the government decided to switch to cost plus fixed fee (CPFF) contracting. In CPFF, all development costs are borne by the government, and the contractor is guaranteed an additional fee above the development costs. In this arrangement, the government assumes all the risks. In order to execute these programs faster, it was decided to award development and production contracts based on requirements rather than on prototypes. In order to ensure that the government's interests were being protected from fraud, waste and abuse, the requirements

evolved to be more detailed and tracing the requirements into the design became more important. So we come to the important observation:

> Modern systems engineering dependence on detailed requirements traces back to a decision to move from contractors building prototypes under fixed-price contracts to the government assuming the risk and awarding contracts based on requirements. The reliance on detailed requirements grew over the years.

A big boost to cost plus fixed fee contracting was provided by the Congressional Commission (Young, 2000) called to investigate a series of costly space program failures during the NASA "Faster, Better, Cheaper[9] (FBC)" period spanning 1992–1999. The study diagnosed that FBC removed much of government oversight, made prior mandatory standards optional, and permitted contractors to cut systems engineering efforts as well as many tests. These cuts, according to the study, led to the failures. The report recommended a stronger role of systems engineering in programs, a return to more government oversight and standards and testing as you *fly*.[10] It also recommended more relaxed budgeting and cost-plus contracting. These recommendations were adopted, and the number of program failures decreased dramatically (Oppenheim, 2011). Regretfully, the technical successes in the post-FBC period were not accompanied by business successes. Abundant evidence indicates that many recent programs have experienced severe budget and schedule overruns, and some programs have been closed for non-performance (see **Table 1**).

The less-than-perfect performance of aerospace programs led the government to make requirements even more detailed, hoping that a tighter definition would yield better execution. The number of Level-2 (system level) requirements grew. Political and financial pressures from numerous contractors and subcontractors led government program offices to impose specific design elements on the system, which again increased the number of requirements. Risk-adverse government agencies also mandated a large number of standards to be followed by the contractors and subcontractors—both technical and programmatic. With the complexity of the requirements and systems growing, customer program offices imposed additional programmatic and management requirements, such as prescribing detailed reporting and stakeholder interactions. As a result, it is not unusual to see recent aerospace programs burdened with as many as several thousands of system level (Level 2) requirements. On average, each of the technical requirements is allocated to about ten lower-level requirements. Anecdotal opinions of practitioners abound that such programs tend to be un-executable. Indeed, a number of large aerospace programs have been terminated (e.g., SIBRSS, NPOESS), and others are notorious for exceeding budgets and schedules (e.g., F-35 fighter).

2-2. The myth of definable interfaces

NASA's SE Handbook (2007) states:

> The bulk of integration problems arise from unknown or uncontrolled aspects of interfaces. Therefore, system and subsystem interfaces are specified as early as possible in the development effort. Interface specifications address logical, physical, electrical, mechanical, human, and environmental parameters, as appropriate…Interface specifications are verified against interface requirements…In verifying the interfaces, the system engineer must ensure that the interfaces of each element of the system or subsystem are controlled and known to the developers (p. 82).

With "n" elements in the system, there are n(n-1)/2 possible connections or interfaces. A typical space vehicle or craft has tens of thousands of elements. This alone makes the interface definition effort formidable, as each interface is needed to write good specifications. Systems engineering practitioners anticipating interfaces understand the trepidation question: "Have we included all of them?" knowing that even one omitted interface may cause fatal failure.

Particularly challenging are the interfaces involving humans. Armstrong (2014) stated: "Human beings are naturally wicked; therefore, interfaces with humans are inherently wicked."[11]

In addition, most of the interfaces traditionally analyzed in technical systems are of the first order, with second- and higher-order effects poorly understood and ignored. Following are examples of well-known failures involving ignored human element and higher-order effects.

1. The Shuttle Challenger Disaster: Engineers understood that the rubber O-rings in the solid motor boosters must not be used in cold weather. They ignored the second-order human interface between the O-ring and the shuttle flight management. The managers did not appreciate the risk of cold weather and ordered the flight, which led to the catastrophe (Challenger Commission, 1986).

2. The Shuttle Columbia Disaster: The interface between the foam covering the cryogenic tank and the airflow, as well as the secondary effect of the foam hitting and damaging the orbiter wings were poorly understood and ignored. The subsequent investigation determined that "the foam did it, the culture allowed it" (CAIB, 2003).

Both above interfaces involving "management" and "culture" qualify as "wicked" human interfaces.

Model-based systems engineering (MBSE), with its advanced functional and physical views of the system being created, greatly facilitates the management of interfaces, but cannot assure that all interfaces have been included, particularly the "wicked" ones. Model-based systems engineering can help in identifying possible interfaces by making the n-squared matrix easier to manage, but cannot fill in the details in each matrix cell. That task is still left to the experience and intuition of engineers. The problem is that the experience and intuition work well only for well-understood systems. This brings us to the following important conclusion:

It is not realistic that all interfaces in a complex system can be anticipated and defined early in the program. Since all interfaces need to be defined in order to write a complete set of requirements, it follows that it is not realistic to develop good, detailed requirements at the program beginning.

2-3. Growing emphasis on flow down of requirements and subcontracting as obstacles to efficiency

Possibly the most critical factors diminishing the efficiency of recent programs are the following two widespread industrial practices:

1. Distribution of production among as many contractor sites and supplier locations as possible, driven by political and financial pressures to "spread the wealth" and secure broad political support for the project.

2. Overwhelmingly popular corporate policy to "stick to core competencies and subcontract the rest," with the vast majority of system parts built by a complex network of suppliers. This complex multi-tier supplier network is largely driven by the fact that higher-tier costs are higher than those at a lower tier, and the time between building parts at higher tiers is so long that many organizations decide to outsource lower-level design activities to specialty contractors who survive because they focus on just one type of activity for multiple buyers.

The heavily outsourced and geographically distributed program makes the program coordination and system integration challenging and increases the need for excellent classical systems engineering and program management. More specifically, prime contractors of modern weapons perform system design, major structural design and systems integration, and subcontract subsystems and components to the established vendor base. Thus a countrywide, distributed tier system has developed in the United States, with prime contractors subcontracting to subsystem providers for major subsystems (e.g., engines, avionics). These second-tier subcontractors then subcontract for components (such as valves, pumps, etc.). These third-tier contractors subcontract for smaller parts such as sensors, actuators, seals, and so forth. It is normal for a supplier network that builds a complex system to include four tiers of suppliers.[12] **Figure 1** illustrates notionally this outsourcing structure, shown here simplified by orders of magnitude.

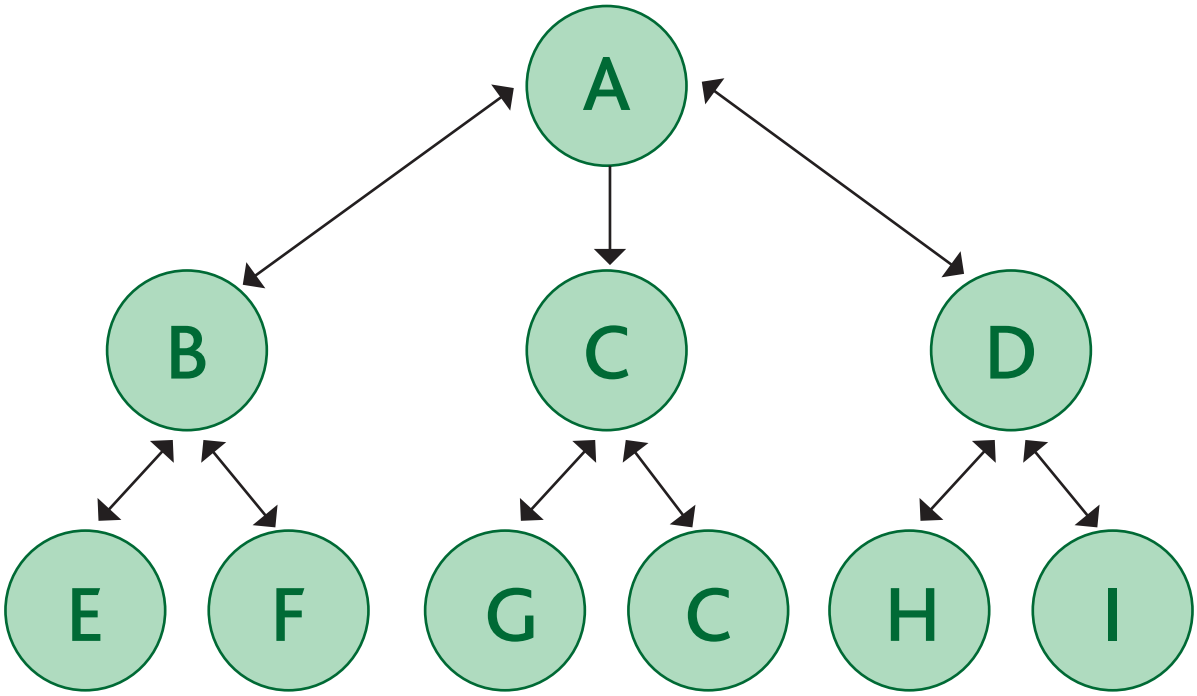

Figure 1: Supplier network.

In **Figure 1**, the prime contractor or government entity "A" divides a program into three projects managed by companies B, C, and D, which in turn subcontract their efforts to E, F, G, H, and I.

As Denning (2013) points out, in a structure such as this, contracts, requirements, and specifications perform a critical role since the performance of the prime contractor can never exceed the capabilities of the *least* proficient of the suppliers. Requirements and specifications prescribe the technical performance and interfaces between multiple parties. Typically, fixed-price contracts define the budgets and schedule for each supplier.

Usually, under such contracts, cost and schedule impacts of a requirement change are absorbed by a higher-level supplier, but cost and schedule impact due to difficulties in achieving technical performance tend to be borne by the supplier unless the buyer has changed the specifications. Typically, this association of requirements with cost and schedule is a large activity of program management.

As a large federal program, Apollo created significant political advantages to producing parts in as many states as possible (Johnson, 2002). This trend followed in the Space Shuttle program; it went totally global for the International Space Station (ISS) program with 15 contributing countries, and for the Boeing 787 program with parts produced in a multitude of countries. In large, modern programs, as much as 70% to 90% or more of value is outsourced, and the total number of companies involved in all tiers reaches tens of thousands (Denning, 2013).

With such a distributed network of production, all linked by legal and financial contracts, the only way to effectively produce systems is to develop excellent top-level requirements, then flow down and allocate them into subsystem requirements, which then flow down and allocate these requirements into component requirements and "build-to" specifications.

In this rigid system of thousands of contracts with different suppliers in all tiers, any change of top-level requirements must be flowed down into all relevant suppliers. As a result, contracts must be re-negotiated, typically with large delays, cost growth or compromised performance in system development. Unstable requirements are the notorious cause of arguments and legal actions between buyers and suppliers in the supply networks.

The huge problem present in practically all programs creating complex systems is that the knowledge about the system, which is necessary to define the top-level requirements and their allocations, is not available until both the system and program are quite mature. This leads classical systems engineering and program management to the following "Faustian Bargain:"

> **Faustian Bargain**
> Either develop and anticipate all interfaces and requirements early, when the knowledge is not yet available, hoping to make the systems engineering Vee process in one pass and to feed the requirements to numerous suppliers early; or conduct massive, painful, and cost- and schedule-busting requirements changes throughout the program, renegotiating the requirements with the suppliers.

Layers of requirements have become a major systems engineering tool for integrating the parts into a desired system for geographically dispersed multi-tiered suppliers. **Figure 2** presents the classical Vee process of systems engineering. Good systems engineering practice requires that each requirement be written together with its verification means—test, analysis, observation, or demonstration. After components are built, they are verified and then integrated into subsystems and verified, and finally integrated at the system level and verified. The requirement flow down is illustrated along the left side of the Vee and the verification along the right side. The Vee process is usually performed with advanced tools for requirements management. Together, the requirements management portion of a complex program often constitutes at least half of the overall classical systems engineering effort.

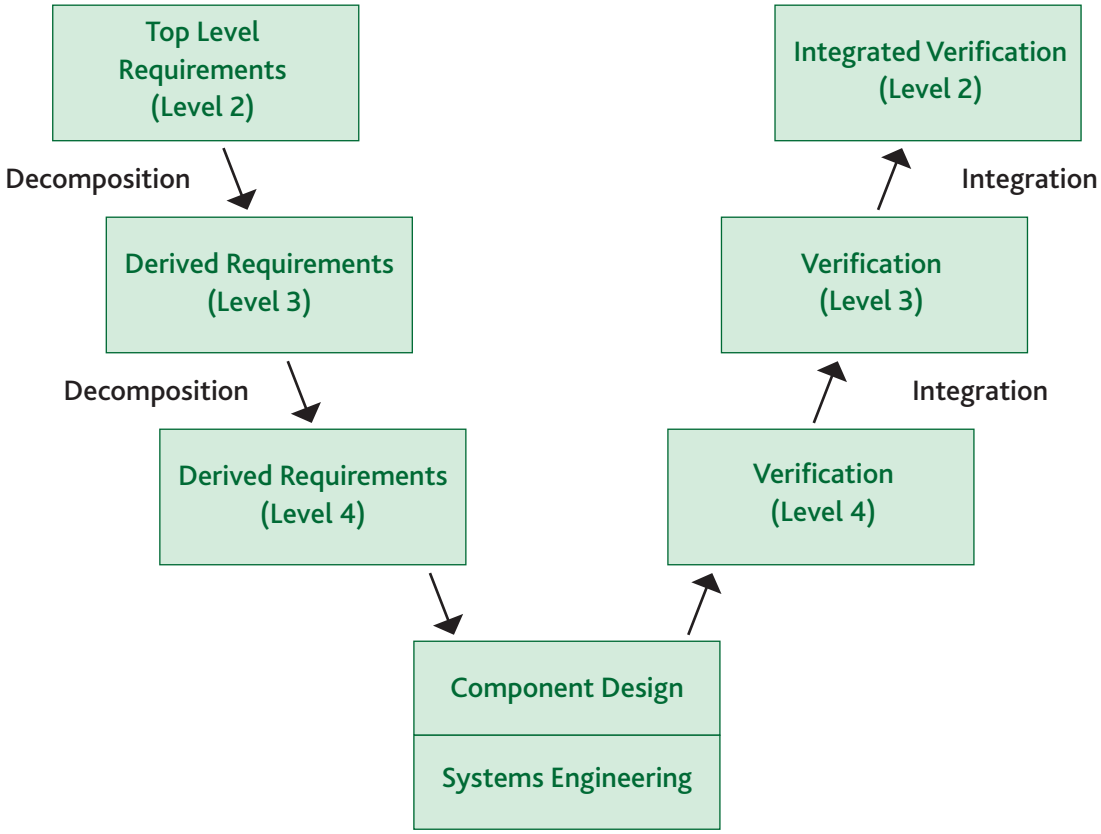

Figure 2: Classical systems engineering Vee (adapted from INCOSE, 2011, p. 27).

Requirements are changed because the knowledge needed to formulate good requirements is not available when they are initially documented. Great engineering must allow for frequent opportunities for creative system- and subsystem-level improvements and optimizations. But, in this horizontally integrated enterprise, improvements and optimizations usually require changes in numerous requirements to numerous subcontractors. One small optimization change at the system level may trickle down into hundreds or even thousands of subcontracts requiring reworking of contracts, requirements, costs, and schedules. Because the disturbance and overall cost of this reworking of contracts tend to be severe, they usually occur in traditional programs only when the program hits a major issue. Special "fire-fighting" teams are then called in to re-baseline and correct the design. In the process, many original requirements are changed, usually with a serious impact on program cost, schedule, and even system performance. As a consequence, program management avoids such program changes as much as possible. Subcontractors working in isolation behind specifications and contractual barriers tend not to do the kind of communication that encourages changes and optimizations. They prefer to solve their own problems of meeting a specification in isolation and are reluctant to participate in multi-vendor solutions or optimizations. A consequence of this fact is that large, complex systems are rarely optimized, and in extreme cases may place the system at serious risk. The sidebar, *Space shuttle center-of-gravity integration problem*, provides a profound example of this culture.

Space shuttle center-of-gravity integration problem

Bob Ryan, former director of the Structures Laboratory of the Marshall Space Flight Center relates how the failure to conduct multi-element optimization significantly inhibited the success of the Space Shuttle program (Covert, 1978). Ryan points out that early in the space shuttle orbiter development, there was a concern that the main engines in the back of the vehicle would shift the center of gravity of the vehicle so far aft that the orbiter would be unstable when in gliding flight. As such, the space shuttle main engine project was given a strict and challenging weight allocation, especially given the challenging thrust requirements for this advanced engine. Marshall Space Flight Center and its contractor, Rocketdyne, successfully designed a powerful engine that met the weight targets. In order to accomplish this goal, the engine designers had to optimize and minimize weight wherever they could. As a result, as the engine went into test, there were a number of catastrophic and expensive failures. Adding weight back into certain areas of the engine to improve structural margins in order to eliminate the catastrophic failure mode was the only way to solve those failures. It also required re-engineering other parts of the engine to make the weight available to solve the original problem. This was a difficult and expensive process that took quite a while to complete, with each catastrophic engine test significantly delaying the program.

The irony of this is that as the design of the orbiter progressed, it turned out that the orbiter had a forward center-of-gravity problem due to the heavy weight of the crew cabin. Although an aircraft with a center of gravity too far aft may be unstable, one with a center of gravity too far forward is so stable that it becomes uncontrollable. For example, it cannot be steered responsively to a new desired flight path. Factors of safety in the cabin design finally had to be reduced from 1.5 to 1.4 to reduce the weight of the crew cabin. In the end, ballast had to be added in the aft of the vehicle to bring the center of gravity back far enough so that the vehicle would be sufficiently aerodynamically controllable. So the engine development was pushed to unreasonable levels of weight reduction to keep the center of gravity forward while at the same time the orbiter crew cabin was being stripped in order to move the center of gravity aft!

Knowledge of this issue finally surfaced about a year before the first shuttle flight, but by that point, the damage had already been done. Throughout the life of the program, the shuttle would pay three penalties for this issue. First, it had to bear a performance penalty due to carrying ballast to move the center of gravity aft. Second, the orbiter crew cabin operated at a reduced factor of safety. Third, the engines operated with reduced margins throughout the design. This example represents a painful testimony to how early allocation of requirements and specifications can inhibit system optimization (Scallion & Phillips, 1985).

Hart-Smith (2001)[13] presented two powerful and well-substantiated arguments against massive outsourcing. First, he documented the fact that outsourcing tends to outsource profits from a prime contractor to its subcontractors because subcontractors operate under fixed-price contracts. As a result, the prime contractor has to absorb the costs of any design and requirement changes. Second, massive outsourcing introduces massive technical problems in system integration. This paper became quite controversial during the Boeing 787 aircraft development, which took the scope of outsourcing to extreme levels and experienced severe schedule and cost consequences. The paper was written before the 787 aircraft program, but served as a "prescient warning to Boeing on 787 trouble"[14] (Reason, 2011).

2-4. The myth of a single-cycle of classical systems engineering

As pointed out earlier, the last 40 years of the U.S. space program demonstrate little success in attempts by the U.S. government and private sector to write a stable set of requirements to design safe, reliable, and cost-effective complex systems. The cited GAO (2011) report states, on average, 82% of the original requirements change for a variety of reasons over the program lifecycle. These issues likely have the same root cause, namely the formal classical systems engineering and program management methodology has evolved based on the critical assumption that knowledge exists early in the program to anticipate all system interfaces. It then performs intensive development of requirements and planning, and subsequently just executes the program in a single cycle of requirements→allocation→ design→build→integrate→verify→validate, as shown by the Vee in **Figure 2**. The INCOSE Systems Engineering Handbook (INCOSE, 2011) provides the following discussion on the Vee:[15,16]

> The Vee model provides a useful illustration of the SE activities during the life-cycle stages. [...] time and system maturity proceed from left to right. The core of the Vee (i.e., those products that have been placed under configuration control) depicts the evolving baseline from user requirements agreement to identification of a system concept to definition of elements that will comprise the final system. With time moving to the right, the evolving baseline defines the left side of the core of the Vee [...] As entities are implemented, verified and integrated; the right side of the core of the Vee is executed. Since one can never go backward in time, all iterations in the Vee are performed on the vertical "time now" line. Upward iterations involve the stakeholders and are the in-process validation activities that ensure that the proposed baselines are acceptable. The downward vertical iterations are the essential off-core opportunity and risk management investigations and actions. In each stage of the system life cycle, the SE processes iterate to ensure that a concept or design is feasible and that the stakeholders remain supportive of the solution as it evolves (p. 27).

In short, classical systems engineering and program management assumes that it is possible to formulate good requirements to deliver extremely complex devices such as satellites, aircraft, refineries, nuclear power plants, and submarines. It also assumes that it is only a matter of providing enough resources to work out the details and quick iterations "on the vertical 'time now'" (INCOSE, 2011, p. 27) to create the needed system.

The reality is that such formal techniques are effective only if designing a commodity for which significant legacy knowledge[17] is available. These techniques break down when the knowledge of what needs to be done to write good requirements is lacking in early program phases, or the requirements are poorly formulated, as discussed below. And the more complex the system, the less knowledge is available just when it is needed. Because changes to specifications, integration and test plan, design and contracts take significant amounts of time, it is very difficult to keep the program baseline consistent. This becomes almost impossible if any program element refuses to accept a change impact from outside its domain due to the cost and schedule impacts. The sidebar, *Space shuttle main engine debris issue*, describes such a painful example in the shuttle program.

Space shuttle main engine debris issue

The shuttle main engines were managed by the Marshall Space Flight Center and developed by the contractor Rocketdyne (now Pratt Whitney Rocketdyne). The Johnson Space Center and its contractor Rockwell (now Boeing) managed the shuttle orbiter. The engine was only tested to handle debris in the propellant systems smaller than 10 microns in diameter. The engine project could not guarantee that an engine explosion would not happen with particles larger than 10 microns due to particle impacts causing local heating and ignition. The orbiter project, which provided the plumbing of the main propulsion system between the external tank and the shuttle main engines, designed the system with a 100-micron filter, 10 times larger than the engine design would guarantee safe. These requirements were fully documented, design certification documents were signed, and the system operated with this major discrepancy for years until this was pointed out in the post Columbia accident investigation. At that point, neither side of the argument was willing to adjust. The orbiter project was concerned about maintaining proper flow rates and pressure if it reduced the filter pore size, and the engine project was unwilling to perform hazardous testing with larger particles. As a result, the system continued throughout its operational life with this discrepancy in requirements and with this hazard unmitigated (Scallion & Phillips, 1985).

Even though the Vee is an elegant and logical approach, it contains poor mechanisms for incorporating new knowledge, requirements changes, or interface changes. When a complex program starts with a large number of detailed requirements (and the recent trend is for increasing numbers), and when the knowledge of what is needed evolves over time, the Vee can be the source of program instabilities involving significant "brute-force" iterations, information churning, and thrashing due to the program pressures to keep requirements and test plans consistent. These factors tend to drive up cost and the duration of complex programs that execute using classical systems engineering and program management. Recently, the cost and time to develop such complex systems reached billions of dollars in decades-long programs.

The recently introduced model-based systems engineering (MBSE) approach (Long & Scott, 2012; Friedenthal, Moore, & Steiner, 2011) addresses these issues by strongly automating and facilitating requirements and interface management—dramatically reducing the time, cost, error rate, and pain of the systems engineering process. While it offers a number of other significant benefits, model-based systems engineering is a tool of classical systems engineering and program management and suffers from the same fundamental problems:

- The inability to formulate stable requirements from the start in complex programs;
- Inability to assure that all interfaces have been captured; and
- Inability to eliminate the huge burden of changing numerous contracts with hundreds or thousands of vendors when a change is needed.

2-5. "Sloppy and careless" requirements

It is unfortunate that data about the requirements instability in large governmental programs are very difficult to acquire (see end note 6). This is because metrics of requirements stability are not used in programs. Some reasons for this are related to national security; some to natural disinclination of stakeholders to "wash the dirty laundry in public"; and some simply due to complex bureaucracy. So, out of necessity, there is only anecdotal knowledge of the programs or limited data from top-level GAO publications. Much of this section is written based on anecdotal knowledge. From the principal investigator perspective, these observations are qualitatively consistent across complex governmental programs in recent years.

Large defense or infrastructural programs often begin with solicitation of goals, needs, and requirements for a new system from a large number of individuals dispersed throughout different government agencies across the nation. The task of formulating requirements for a far-off future system often has a low priority for many of the individuals involved and so this task is often delegated to junior people. The junior staff often lacks the expertise about the new system. Some, who feel under pressure to submit some requirements, simply cut-and-paste requirements from an earlier program. A well-known occurrence of this was noted in a recent satellite program, which included customer requirements that applied only to a submarine (Oppenheim, 2013). Regretfully, anecdotal knowledge indicates that requirements are often carelessly formulated,[18] do not fulfill any purpose in the system, or serve a parochial reason.[19]

All the submitted requirements become a part of a document that serves as the basis for a request for proposal (RFP) or sole-source contract. As mentioned above, in recent complex space programs the total set of customer-level technical and non-technical requirements is measured in the low thousands. This set includes the best efforts in the development of technical requirements, as well as the requirements that should not have been included. Among the latter are the requirements that:

- Do not contribute to value/benefit;
- Are sloppy, unclear, or incorrectly formulated;
- Are mutually conflicted;
- Are parochially motivated;
- Represent "gold-plated" requirements; and
- Mandate that a particular subsystem design be included in the system, thus making system optimization difficult.

In addition, programmatic requirements intended to apply tight controls over the program execution are included, but these requirements often only increase program bureaucracy, cost, and time. With this large number of requirements, often the most difficult ones—those dealing with wicked or higher-order interfaces—are missing simply because the system is too complex to provide good insight early in the program.

Despite the fact that Federal Acquisition Regulations (GSA, 2005, Part 7.105) direct federal agencies to engage with industry in the development of multiple iterations of proposals, it simply isn't done. Crowdsourcing requirements[20] in partnership with industry as part of the proposal process would help the government keep pace with available technologies. This is a cultural barrier that must be overcome, particularly given the pace of technological advances, evolution of innovative niche technology firms/start-ups, etc., which can affect defense systems and engineering programs.

Ideally, a rigorous independent review of all requirements should be performed after they are collected and before the requirements are released for RFP or contract. Highly competent and independent reviewers should be able to catch all instances of unneeded and faulty requirements, missing interfaces, and "gold plated" requirements. The reviewers should be able to demand that all these deficiencies be corrected before the program proceeds to the RFP or contract. Unfortunately, such reviews are hardly ever performed. The net result is that large, complex programs start with bad requirements, which, of course, cause significant program waste and penalties in program cost and schedule. In extreme cases, programs are terminated after expending significant treasure.

It also would be ideal if a single individual were given the full responsibility, authority, and accountably (RAA) for the development of all customer-level requirements. Again, this is not the present classical systems engineering and program management practice and represents a major deficiency that is easily correctable. Of course, a single individual cannot possibly have expertise spanning all technical and business areas of a complex program, and must be given sufficient assistance from across the experts relevant to the program scope. Still, a single individual must have the proverbial sign on his or her desk that states: "the buck stops here." Such an individual would make his or her best effort to assure that only the best possible requirements are passed through to the program RFP or contract milestone. That alone would likely prevent many, but not all, subsequent program problems.

3. The Revolutionary SpaceX Business Model

Section 2 of this paper describes the notorious problems plaguing recent complex programs. The root cause of these problems is the inability to define all interfaces and detailed requirements early in such programs when the required knowledge is not yet available. And detailed requirements are needed for flowing down and allocating the requirements to all levels of subsystems in order to sign contracts with suppliers in the entire supply network. The poor-quality requirements defined and allocated prematurely then cause massive program iterations and cost and schedule penalties.

Only a radical change of business model would be able to cure all these problems all at once, such as that practiced by SpaceX. Oppenheim and Muratore (2015) describe this revolutionary company making Falcon rockets and Dragon spacecraft. This company is almost totally co-located and vertically integrated, which is unique in modern space business; it buys raw materials and makes practically all parts, subsystems, and systems in-house (electronics, avionics, software, rocket engines, rockets and spacecraft). Since it has few component suppliers and no suppliers above the component level, there is no need to allocate requirements early in the program when the knowledge to do so is not yet available. SpaceX buys components either by specifications or out of catalogs and then extensively tests them for qualification and acceptance in the intended design application to provide mission assurance. Since there are no lower-level requirements to deal with, there is no need to spend program time and resources on their verification or control boards. There is no need for massive interface control bureaucracy. SpaceX formally verifies only top-level (customer) requirements dealing with the payload and payload interface, and these are relatively easy to define and verify. Instead of a massive management of lower-level requirements, SpaceX devotes the intellectual energy toward system and subsystem optimization using the best engineering and mission assurance methods available: testing "what you fly," including testing of the fully integrated system, and rapid and efficient developmental design–build–test loops.[21] The SpaceX revolutionary model leads to dramatic savings in program schedule and cost, a much better system because of the massive engineering optimization, and the ability to deliver a system at a highly competitive fixed price[22] (Oppenheim & Muratore, 2015).

Since its inception in 2002, SpaceX has developed 11 major flight-proven products in 12 years, including four engines (Merlin, Merlin Vacuum, Kestrel, and Draco Thruster); launch vehicles (Falcon 1, Falcon 9); and Dragon v1 and Dragon v2 spacecraft, along with the associated ground test, launch, and mission facilities. At the time of this writing, SpaceX has demonstrated 17 successful launches and is completing development of the Falcon 9 Heavy vehicle, a reusable, propulsively landing first stage and a reusable, propulsively landing Dragon v2 spacecraft. It signed contracts with NASA and the U.S. Air Force for cargo and crew deliveries to space. The company employs only 4,000 people, the vast majority of them working in one hangar in Hawthorne, California, USA. Given the small number of employees, the short development times, and high technical quality of its products, these are dazzling accomplishments, vastly superior to any existing competition to the knowledge of the author.

The Lean enablers (LEs) (Oehmen, 2012) were not developed to deal with such a radical business model change. A new set of enablers would be needed to capture the SpaceX business model. The following paragraph is a proposed guideline for the development of such new enablers.

> Do not assume that you are capable of anticipating all interfaces and developing good detailed subsystem requirements early in the program. In complex programs it is not realistic because the knowledge required to do so does not exist yet. Especially easy to ignore are the wicked interfaces that involve humans, or second-order interactions. Thus, it is not realistic to expect to be able to develop perfect requirements at the program beginning, and to allocate them to all lower-level subsystems and define all verifications for the benefit of a massive supply network that the business case requires be defined early in the program. Instead, remain vertically integrated and fully co-located. Focus on system optimization rather than lower-level requirements. Implement bureaucracy-free great engineering and mission assurance to optimize your system and subsystems to the best they can be and do not waste time on verification of imperfect lower-level requirements. In order to enable this engineering approach, you will need to create a first-rate, co-located, rapid design–build–test infrastructure to permit frequent, extensive, and efficient prototype tests. Perform the best available mission assurance by continuously testing subsystems and full systems in the "what you fly" condition, and implement technologies to enable testing after vehicle integration. Build the infrastructure to be able to perform such tests efficiently.

4. The Lean Enablers

Dramatically higher performance, like that achieved by SpaceX, involves revolutionary changes to company and program structure, technologies, operations, and testing infrastructure. However, organizations that are not ready for such a radical paradigm shift can utilize incremental changes, such as those contained in *The Guide to Lean Enablers for Managing Engineering Programs* (Oehmen, 2012), to address some of the challenges associated with requirements and outsourcing. The Lean enablers aim at helping more traditional programs better integrate systems engineering and program management, and create better program value and benefit while reducing waste.

In the following text we explicitly list applicable major Lean enablers from *The Guide to Lean Enablers for Managing Engineering Programs* (Oehmen, 2012) that will help various deficiencies in the development and use of program requirements in all programs. These Lean enablers will have a profound effect on the quality of requirements, but they will not have the power to enable a single-pass through the systems engineering Vee. Still, these Lean enablers represent an intermediate step for performance improvement. Some changes of requirements will be inevitable as the program iterates its technical solution.

Co-location is by far the most effective means enabling efficient and real-time, wide-bandwidth communication, and coordination among program stakeholders. The significant benefits of co-location are promoted by the following Lean enablers:

1.6.1 Prefer physical team colocation to the virtual colocation.

4.4.3 Maximize co-location opportunities for program management, systems engineering, business leadership and other teams to enable constant close coordination, and resolve all responsibility, communication, interface and decision-making issues up-front early in the program.

4.9.6 Use Lean tools to promote the flow of information and minimize handoffs. Implement small batch sizes of information, low information in inventory, low number of concurrent tasks per employee, small task times, wide-communication bandwidth, standardization, work cells, and training.

As discussed in the Section 2, since modern programs rely heavily on outsourcing, premature formulation of detailed interfaces and requirements and requirement allocations can be deadly to program efficiency. Therefore the Lean enabler 3.7.1 advises to delay the time for outsourcing as late as possible, until enough knowledge about the system and subsystems has been developed. If followed, this Lean enabler alone would reduce the massive effort of requirements changes and the cost and effort of changing the supplier contracts accordingly.

Healthy implementation of Lean enabler 3.7.1 requires excellent relationships with key suppliers based on seamless partnerships. In practice, this means that relationships with suppliers must be established early and should preferably be intended as long-term partnerships. Detailed, specific requirements should be passed on to these long-term supplier partners only after the requirements are mature and stable. Until they are mature and stable, design and systems engineering should ideally be conducted in a co-located single organization. Since suppliers will now receive their allocated requirements after some delay, the suppliers must be sufficiently lean to deliver the contracted items with the shortest possible lead times. This requires heavy involvement of key

suppliers in program planning and early development, as stated in Lean enabler 3.5.15: Heavily involve the key suppliers in program planning and at the early phases of program.

Vast improvements in the quality of requirements could be achieved by applying the following Lean enablers:

2.1.3 Develop a robust process to capture, develop, and disseminate customer stakeholder value with extreme clarity.

2.1.4 Proactively resolve potential conflicting stakeholder values and expectations, and seek consensus.

2.4 Develop high-quality program requirements among customer stakeholders before bidding and execution process begins.

2.4.1 Assure that the customer-level requirements defined in the request for proposal (RFP) or contracts are truly representative of the need; stable, complete, crystal clear, deconflicted, free of wasteful specifications, and as simple as possible.

2.4.2 Use only highly experienced people and expert institutions to write program requirements, RFPs and contracts.

2.4.3 If the customer lacks the expertise to develop clear requirements, [the customer should] issue a contract to a proxy organization with towering experience and expertise to sort out and mature the requirements and specifications in the RFP. This proxy must remain accountable for the quality of the requirements, including personal accountability.

2.4.4 Prevent careless insertion of mutually competing and conflicting requirements, excessive number of requirements, standards, and rules to be followed in the program, mindless "cut-and-paste" of requirements from previous programs.

2.4.5 Minimize the total number of requirements. Include only those that are needed to create value to the customer stakeholders.

2.4.6 Insist that a single person is in charge of the entire program requirements to assure consistency and efficiency throughout.

2.4.7 Require personal and institutional accountability of the reviewers of requirements until the program success is demonstrated.

2.4.8 Always clearly link requirements to specific customer stakeholder needs and trace requirements from this top level to bottom level.

2.4.9 Peer review requirements among stakeholders to ensure consensus validity and absence of conflicts.

2.4.10 Require an independent mandatory review of the program requirements, concept of operation, and other relevant specifications of value for clarity, lack of ambiguity, lack of conflicts, stability, completeness, and general readiness for contracting and effective program execution.

2.4.11 Clearly articulate the top-level objectives, value, program benefits and functional requirements before formal requirements or a request for proposal is issued.

2.4.12 Use a clear decision gate that reviews the maturity of requirements, the trade-offs between top-level objectives, as well as the level of remaining requirements risks before detailed formal requirements or a request for proposal is issued.

2.5.2 Follow up written requirements with verbal clarification of context and expectations to ensure mutual understanding and agreement. Keep the records in writing, share the discussed items, and do not allow requirements creep.

2.5.3 Use architectural methods and modeling to create a standard program system representation (3D integrated CAE toolset, mockups, prototypes, models, simulations, and software design tools) that allow interactions with customers and other stakeholders as the best means of drawing out requirements.

2.5.4 Listen for and capture unspoken customer requirements.

2.5.8 Create effective channels for clarification of requirements (e.g., involving customer stakeholders in program teams).

2.5.9 Fail early and fail often through rapid learning techniques (e.g., prototyping, tests, simulations, digital models, or spiral development).

2.6 Actively minimize the bureaucratic, regulatory, and compliance burden on the program and subprojects.

2.6.1 Strive to minimize and streamline the burden of paperwork for external stakeholders by actively engaging them in the process and clearly articulating and aligning the benefit generated by each report.

3.1.1 Plan to develop only what needs to be developed.

4.6.3 Seek and maintain independent reviews of the program. Assign teams outside of the program to observe and assess the execution and health of the program. Engage non-advocates in review process.

Lean enablers 2.4.6 and 2.4.10 provide powerful and inexpensive means to vastly improving the quality of the requirements early in the program. Regretfully, they are not yet practiced in government programs.

Many of the problems associated with starting a program prematurely with inadequate technology could be avoided if the following Lean enablers were followed:

3.4 Ensure up-front that capabilities exist to deliver program requirements.

3.4.1 Ensure strong corporate, institutional, and personal accountability and personal penalties for "low-balling" the budget, schedule, and risk, and overestimating capabilities (e.g., the technology readiness levels (TRL)) in order to win the contract.

3.4.3 Ensure that planners and cost estimators are held responsible for their estimates during the execution of the program. Minimize the risk of wishful thinking.

3.10 Manage technology readiness levels and protect program from Low-TRL delays and cost overruns.

3.10.1 Create transparency regarding the technology risks and associated cost and schedule risks before large-scale programs are contracted. Issue small contracts to mature critical technologies before starting a large-scale program.

3.10.4 Utilize program management strategies that produce the best balance between technology risk and reward in your program, such as evolutionary acquisition, incremental, or spiral development.

3.10.6 Remove show-stopping research and unproven technology from the critical path of large programs. Issue separate development contracts, staff with colocated experts, and include it in risk mitigation plan. Reexamine for integration into the program after significant progress has been made or defer to future systems.

3.10.7 Provide stable funding for technology development and maturation. This will support a steady, planned pipeline of new technologies to be inserted into the program.

3.10.8 Match technologies to program requirements. Do not exceed program needs by using unnecessarily exquisite technologies ("gold plating").

3.10.11 Utilize independent technical reviews to confirm a capability to deliver and integrate any new technology that could delay the program or cause schedule overruns.

6.1.4 Do not implement any standard purely for achieving any sort of mandated program certification.

5. Summary and Conclusions

The classical systems engineering and program management methodology is based on the assumption that the knowledge to anticipate all interfaces and create good requirements exists at program initiation; it is a matter of working out the details to build extremely complex devices such as satellites, aircraft, refineries, nuclear power plants and high-speed rail. This works well for well-understood systems, but it breaks down when the knowledge of what needs to be done still needs to be discovered, which is the case with most complex systems. In real programs that develop complex systems, the reality is more reminiscent of the "Faustian Bargain:" Either develop and anticipate all interfaces and requirements early, when the knowledge is not yet available, hoping to make the systems engineering Vee process in one pass and to feed the requirements to numerous suppliers early; or conduct massive, painful, and cost- and schedule-busting requirements changes throughout the program, renegotiating the requirements with the suppliers. As described in the text, the average number of requirements changes in large programs is 82%, which indicates that the second path dominates in industry. As described, this Faustian Bargain is the unintended emerging characteristic of the systems engineering evolution during the last 60 years, driven by the geographical distribution of a vast number of suppliers who must be fed requirements early in order to sign all the thousands of contracts in time. Section 2 discusses other myths of modern classical systems engineering and program management, including the assumption that all system interfaces can be anticipated early.

The situation continues to worsen as the complexity of modern systems increases at significant rates. The constant dynamic of need, innovation, and change makes it increasingly improbable that detailed and stable requirements can be developed at a program's initiation. This observation applies not only to space and national security programs, but to a vast array of other government and commercial technology and socio-technological programs, such as:

- Cyber security systems that must operate effectively without interfering with overall performance of the system;
- Finance;
- Internet communication;
- Energy;
- Nuclear waste disposal;
- Education;
- Global warming;
- Transportation;
- Medical systems; and
- Many others.

In many of these programs, the rational approach is to delay subcontracting until the system design is mature and optimized, and the requirements are stable. Some Lean enablers discussed in this paper strongly favor such a delay.

The high cost and schedule penalty of renegotiating subcontracts in order to accommodate changing requirements and system optimization are not the only arguments against massive subcontracting. As Hart-Smith (2001) discussed from the perspective of a prime contractor, two other reasons are that massive

outsourcing of value creation to numerous suppliers tends to outsource profits from a prime contractor to its suppliers. This practice also introduces massive problems in system integration by the prime contractor.

The SpaceX business model offers a radical elimination of all of the above problems at once. The company is totally vertically integrated and co-located. It does not allocate requirements to subcontractors and does not need to verify the requirements because there are no subcontractors. Instead of a vast bureaucracy of interface control and requirements management, the company's intellectual energy is spent on system optimization and testing, including rapid prototype testing and testing integrated systems in the "what you fly" condition. This approach yields dramatically better systems, as well as shorter and less costly programs. A guideline for the development of additional Lean enablers promoting the SpaceX model has been formulated.

This radical change is just not practical for large governmental programs that have to operate under complex political and financial constraints. For programs of that type, *The Guide to Lean Enablers for Managing Engineering Programs* (Oehmen, 2012) has been published. This paper explicitly listed the Lean enablers that will improve various existing deficiencies in the development and use of program requirements in all programs. These Lean enablers will have a profound effect on the quality of requirements and program execution, but they will not have the power to enable a single-pass through the systems engineering Vee. Some changes of requirements will be inevitable as the program iterates its technical solution. However, compared to the performance of recent complex programs, the Lean enablers should provide a dramatic improvement in program cost, schedule, and quality.

6. Notes

1 ^ The PMI-INCOSE-MIT joint project followed and benefitted from an earlier, similarly large project conducted by the Lean Systems Engineering (LSE) Working Group of INCOSE from 2006 to 2010. The earlier project developed 147 best practices (Oppenheim, 2011). The author of this paper served as co-leader of that project (Oppenheim, Murman, & Secor, 2011). All 147 Lean enablers for systems engineering were incorporated into *The Guide to Lean Enablers for Managing Engineering Programs* (Oehmen, 2012).

2 ^ A Nunn-McCurdy review is a mandatory congressional review for viability of a program triggered at 15% growth over original cost. Growth at 25% over original cost, or 50% over objective, can lead to program cancellation unless deemed necessary for national security.

3 ^ The principal investigator studied the effect of contracting a program before it was ready from the technology readiness point of view (Oppenheim, 2013). Programs are not supposed to proceed to the so-called Milestone B (design) unless all technology items are at Level 6–7 or higher. Yet, for expediency reasons, many programs are contracted with immature technology readiness level.

4 ^ GAO (2014) has reviewed selected, major NASA programs. The report indicated, "NASA projects have continued to make progress in maturing technologies prior to the preliminary design review. This year, 63 percent of projects met this standard, up from only 29 percent of projects in 2010." The report documents significant improvement in program execution due to the use of mature technologies.

5 ^ "Gold plating" refers to the program or system features and options that are excessive and unnecessary.

6 ^ At the beginning of this study, a high-level manager from one of the Federally Funded Research and Development Centers supporting the U.S. Air Force Space and Missile Command offered staff time and budget to extract statistics on stability of requirements over the program lifecycle in major government space programs. After months of trying, it turned out such metrics are not collected and cannot be obtained without major effort.

7 ^ Si Ramo once told Brown (personal communication, 2009) that the first use of systems engineering in modern times could be said to have started at AT&T when they faced assembling a world-wide telephone system. AT&T, however, did not consider it systems engineering and did not use that name.

8 ^ The classical engineering education curriculum does not leave much time for integrative projects, with the majority of the courses devoted to individual and usually disconnected engineering subjects (strength, materials, dynamics, thermodynamics, etc.)

9 ^ The U.S. Department of Defense followed the same philosophy, but replaced the term *FBC* with *Acquisition Reform*.

10 ^ As you fly means using identical (but not necessarily the same) hardware and software to the one to be used in flight. *What you fly* means using the same hardware and software.

11 ^ In this context, "wicked" refers to "unpredictable, mischievous."

12 ^ The common term used in supply chain management to describe a company that has this type of a large chain of suppliers is "horizontally integrated." A company that produces most parts in-house is called "vertically integrated." These terms will be used in subsequent text.

13 ^ The paper is labeled "Boeing Proprietary." However, it was leaked and published on the Seattle Times webpage http://seattletimes.nwsource.com/ABPub/2011/02/04/2014130646.pdf, and, therefore, now exists in the public domain.

14 ^ The following quotes are from the Seattle Times article. "The (Hart-Smith) paper laid out the extreme risks of outsourcing core technology and predicted it would bring massive additional costs and require Boeing to buy out partners who could not perform." The Seattle Times article quotes Boeing Commercial Airplanes Chief Jim Albaugh publically talking about the lessons learned from the disastrous three years of delays on the 787 Dreamliner. "…the 787's global outsourcing strategy—specifically intended to slash Boeing's costs—backfired completely." "Albaugh said in the interview that he read the (Hart-Smith) paper six or seven years ago, and conceded that it had "a lot of good points" and was "pretty prescient." "'We spent a lot more money in trying to recover than we ever would have spent if we'd tried to keep the key technologies closer to home,' Albaugh told his large audience of students and faculty." "Boeing was forced to compensate, support or buy out the partners it brought in to share the cost of the new jet's development, and now bears the brunt of additional costs due to the delays. Some Wall Street analysts estimate those added costs at between $12 billion and $18 billion, on top of the $5 billion Boeing originally planned to invest."

15 ^ The appearance of the INCOSE Handbook Vee is slightly different from our Figure 1, but illustrates essentially identical information.

16 ^ Other popular systems engineering manuals, such as NASA (2007), (SMC, 2011) essentially follow the same timeline logic.

17 ^ For example, in designing a "routine" missile it is relatively easy to define a warhead weight, range, and accuracy requirements. From that point, requirements can be derived and allocated in one pass.

18 ^ In order to be effective, a requirement must be carefully, clearly, and unambiguously formulated using precise language and technical rules.

19 ^ An example of a parochial reason is a not-unusual imposition of a specific design element to be included in the system for no other reason than a preference of a customer stakeholder.

20 ^ Crowdsourcing is an emerging, typically online, distributed problem solving and production model where a problem is solved through the involvement of a large number of people (Hosseini et al., n.d.).

21 ^ This brief explanation does not do justice to the SpaceX process. Interested readers should read Oppenheim and Muratore (2015).

22 ^ In September 2014, NASA signed contracts with both SpaceX and Boeing for space vehicles transporting crews to the International Space Station. The price of the SpaceX vehicle is less than half of the price of the Boeing vehicle.

7. References

Armstrong, J. R. (2014). System integration: He who hesitates is lost. *INCOSE International Symposium*. Las Vegas, 2014.

Challenger Commission. (1986). *Report to the President by the Presidential Commission on the Space Shuttle Challenger Accident.* Retrieved from http://history.nasa.gov/rogersrep/genindex.htm

Columbia Accident Investigation Board (CAIB). (2003). *Report Volume 1*. Retrieved from http://s3.amazonaws.com/akamai.netstorage/anon.nasa-global/CAIB/CAIB_lowres_full.pdf.

Covert, E. E. (1978). Technical status of the space shuttle main engine: A report of the ad hoc committee for review of the space shuttle main engine development. Washington, DC: Assembly of Engineering, National Research Council, National Academy of Sciences, March 1978.

Denning, S. (2013). What went wrong at Boeing? *Forbes*. Retrieved from http://www.forbes.com/sites/stevedenning/2013/01/21/what-went-wrong-at-boeing.

Friedenthal, S., Moore, A., & Steiner, R. (2011). *Practical guide to SysML, Second edition: The systems modeling language*. Burlington, MA: Morgan Kaufmann.

General Services Administration (GSA). (2005). Title 48—Federal Acquisition Regulations System, Chapter 1, Federal Acquisition Regulation. Effective 2 March 2015. Retrieved from http://www.acquisition.gov/far/current/pdf/FAR.pdf.

Government Accountability Office (GAO). (2011). *Defense acquisitions: Assessments of selected weapon programs*. Retrieved from http://www.gao.gov/new.items/d11233sp.pdf.

Government Accountability Office (GAO). (2014). *NASA: Assessments of selected large-scale projects*. Retrieved from http://www.gao.gov/assets/670/662571.pdf.

Hart-Smith, L. J. (2001). Outsourced profits—The cornerstone of successful subcontracting. *Boeing Third Annual Technical Excellence (TATE) Symposium*. St. Louis, Missouri, February 14–15, 2001. Retrieved from http://seattletimes.nwsource.com/ABPub/2011/02/04/2014130646.pdf.

Hosseini, M., Phalp, K., Taylor, J., & Ali R. (n.d.). *Towards crowdsourcing for requirements engineering*. Retrieved from http://ceur-ws.org/Vol-1138/et2.pdf.

International Council on Systems Engineering (INCOSE). (2011). *Systems Engineering Handbook*, Version 3.2.2. San Diego, CA: Author.

Jacobson, T. C. (2001). TRW 1901-2001: *A tradition of innovation*. Cleveland, OH: TRW Inc.

Johnson, S. (2002). *Secrets of Apollo: Systems management in American and European space programs*. Baltimore, MD: Johns Hopkins University Press.

Long, D., & Scott, Z. (2012). *A primer for model-based systems engineering*. Blacksburg, VA: Vitech Corporation.

National Aeronautics and Space Administration (NASA). (2007). *NASA Systems Engineering Handbook, NASA/SP-2007-6105 Rev 1*. Retrieved from http://www.acq.osd.mil/se/docs/NASA-SP-2007-6105-Rev-1-Final-31Dec2007.pdf.

Oehmen, J. (Ed.). (2012). *The guide to lean enablers for managing engineering programs*, Version 1.0. Cambridge, MA: Joint MIT-PMI-INCOSE Community of Practice on Lead in Program Management. URI: http://hdl.handle.net/1721.1/70495.

Oppenheim, B. W. (2011). *Lean for systems engineering with lean enablers for systems engineering*. Hoboken, NJ: Wiley & Sons.

Oppenheim, B. W. (2013). Improving affordability: Separating research from development and from design in complex programs. *Crosstalk Journal, 26*(4).

Oppenheim, B. W., & Felbur, M. (2014). *Lean for banks: Improving quality, productivity, and morale in financial offices*. New York, NY: CRC Press.

Oppenheim, B. W., & Muratore, J. (2015). Systems engineering at SpaceX, submitted to *Journal of Systems Engineering*.

Oppenheim, B. W., Murman, E. M., & Secor, D. A. (2011). Lean enablers for systems engineering, *Systems Engineering. 14*(1), 29–55. DOI: 10.1002/sys.20161.

Pabst, R. G. (2014). *Analysis of management practices in U.S. space programs using GAO data, and mapping to Lean enablers*. Systems Engineering Capstone Project, SELP Capstone Project. Loyola Marymount University, Los Angeles, CA.

Reason, J. (2011). The contribution of latent human failures to the breakdown of complex systems. Phil. Trans. R. Soc. Lond. B. 12 April 1990. Seattle Times, "A prescient warning to Boeing on 787 trouble", February 5, 2011.

Scallion, W. I., & Phillips, W. P. (1985). *Space shuttle orbiter trimmed center-of-gravity extension study*. Retrieved from http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19850010693.pdf.

Space & Missile Systems Center (SMC). (2011). Systems engineering: Specialty engineering disciplines. U.S. Air Force. Retrieved from http://www.everyspec.com.

Young, T. (2000). *Report on Mars program failure*. U.S. Congress, Science Committee. Retrieved from http://orion.asu.edu/Failure%20Reports/Mars_Observer_12_93_MIB.pdf.

Acknowledgments

The author wishes to thank Mr. John Muratore of SpaceX for sharing the material that served as the basis for writing parts of Section 2 of this paper. Ms. Azadeh Bateni, graduate student of Systems Engineering at Loyola Marymount University, Los Angeles, California, USA, has skillfully prepared the illustrations.

About the Author

Bohdan "Bo" W. Oppenheim is a Professor of Systems Engineering at Loyola Marymount University in Los Angeles, California, USA, and founder and co-chair of INCOSE Lean Systems Engineering Working Group, the largest Group of INCOSE. He co-led the project developing Lean Enablers for Systems Engineering (LEfSE), and served as an expert in the joint INCOSE-PMI-MIT project developing Lean Enablers for Managing Engineering Programs, integrating Lean Systems Engineering with Lean Project Management. Dr. Oppenheim authored the book *Lean for Systems Engineering with Lean Enablers for Systems Engineering (Oppenheim, 2011)*, and co-authored *The Guide to Lean Enablers for Managing Systems Engineering Programs (Oehmen, 2012)*, and co-authored Lean for Banks (Oppenheim & Felbur, 2014). He worked for five years at Northrop, four years at Aerospace Corporation, and consulted numerous defense and aerospace companies in the United States and Europe. His engineering degrees include a PhD from Southampton, U.K.; a Graduate Engineer's Degree from MIT; an MS from Stevens Institute of Technology; and a BS (equiv.) from Warsaw University of Technology in Aeronautics. His industrial experience spans space, offshore, mechanical engineering and software, including several major aerospace and commercial programs. His credits include five books, 20 journal publications, US $2.5 million in externally funded grants, and a 30-year industrial experience. He was recently named an INCOSE Fellow. He served on the teams honored with two Shingo Awards (2011 and 2013), the INCOSE Best Product Award (2010) and the INCOSE Service Award (2014).

Beijing | Bengaluru | Brussels | Buenos Aires | Dubai | Lelystad | Mumbai | New Delhi
Philadelphia | Porto Alegre | Rio de Janeiro | Shenzhen | Singapore | Washington, DC

PMI.org

Project Management Institute
Global Operations Center
14 Campus Blvd
Newtown Square, PA 19073-3299 USA
Tel: +1 610 356 4600

Project Management Institute

Making project management indispensable for business results.®