

So It's Complex, Why Do I Care?

Steven Holt, Paul Collopy, and Dianne DeTurris

1 Introduction

The subject of this chapter is systems engineering and how it is impacted by complexity. Systems engineering is a transdisciplinary process that engineering organizations use to develop aircraft, spacecraft, radars and other similar large systems. Prior to systems engineering, an informal process existed for designing engineered objects, and the process worked well for individuals or teams who designed light bulbs, electric motors, and even ocean liners. However, space launch rockets and jet propelled aircraft could not be designed by individuals, or even small teams. These are systems with many components, and different engineering specializations are necessary to design different components. Designing a modern aircraft requires whole organizations of engineers that includes many teams.

Systems engineering seemed effective in the 1960s. Apollo landed on the moon. The Concorde was designed in the late 1960s, and eventually carried passengers at Mach 2 across the Atlantic Ocean. The Boeing 747 and the Mach 3 Lockheed SR-71 were also designed in the late 1960s. Even the sky was no limit.

However, the record of systems engineering achievement from 1990 to the present has been less impressive. Although the Curiosity rover has actively explored Mars and the Hubble Space Telescope has set records for generating

S. Holt (✉)

The Boeing Company, Seattle, WA, USA

e-mail: steven.c.holt@boeing.com

P. Collopy

Department of Industrial & Systems Engineering & Engineering Management, University of Alabama in Huntsville, Huntsville, AL, USA

D. DeTurris

Aerospace Engineering Department, Cal Poly State University, San Luis Obispo, CA, USA

terabytes of scientific data, and even though the safety record of modern jet airliners is the envy of the transportation industry, today's process for developing large engineered systems is problematic. Cost overruns are so common that they are actually expected, and the average overrun is around 50 %. Most projects finish years late, if they finish at all. One third of large US defense projects (from 1990 to 2007) and one half of NASA human space exploration projects (from 1991 to 2011) have been cancelled.

Why has systems engineering become so difficult? The usual explanation is that modern engineered systems are complex. This is descriptively informative, but the practicing systems engineer needs a pragmatic guide to assessing complexity, and based on that assessment, taking an appropriate approach to reduce, mitigate, adapt to, or exploit complexity and bring large projects to successful conclusions.

So, why do I care? One must care whether the system is complex, because complex systems are not predictable. Contemporary systems engineering is founded on requirements, which are essentially predictions of the outcome of the system design and development process. The difficulties in large system development programs today are that essentially, the system fails to meet its requirements. Cost and schedule overruns are mere symptoms, as programs attempt to recover or patch over the gap between what the system was predicted to do and what it actually does. In this sense, the root of system development difficulties is the prediction, via requirements, of the outcome of a process that is fundamentally not predictable.

In this chapter, the management of complexity is explored as a transdisciplinary element of system design. That is, complexity does not belong to mechanical design, software engineering, or human factors. Instead, a design can be characterized as complex or not complex from any or all of these perspectives.

At the same time, management of complexity is not itself a hard science like dynamics or electromagnetics. It is barely quantitative, and perhaps only qualitative. But even as a qualitative measure, appreciation of the complexity of a system provides the design organization with a warning of approaches to avoid and perhaps some positive steps toward more successful outcomes.

For example, this chapter recommends the Cynefin Framework as an approach to system development that will accommodate emergent behaviors which are characteristic of complex systems. Small, safe to fail experiments are suggested for learning about a system that defies the best attempts to predict behavior.

The purpose of this chapter is to develop the beginnings of a prescriptive approach to systems engineering for complex engineered systems. It shows that complexity inherently means system behaviors will occur that cannot be predicted. The standard, rational approach of taking actions that leads to the desired outcomes will not work on complex systems, because it is not clear where actions lead. This inability to predict system attributes and behavior necessitates a re-examination of the fundamentals of systems engineering practice. This re-examination ultimately must entail not only complexity science, but also cognitive science, social psychology, organizational sociology, and the physics of the designed artifact.

1.1 The Rising Cost of Aerospace Systems with Complexity

The design of large engineered systems today is increasingly difficult because of the level of complexity in the systems that are designed. For example, in the spring of 1939, the aircraft manufacturer Consolidated received a contract to develop a prototype aircraft for what became the B-24 Liberator long range bomber. The aircraft began service in 1941. In the fall of 1986, Boeing and Lockheed received contracts to develop competing prototypes for the aircraft eventually designated the F-22 fighter. The aircraft began service in 2005. The progression of development time increased from 2 years to 19 years between these programs. The F-22 development cost is roughly 1000 times the comparable B-24 cost. So what changed? The usual answer is that the F-22 is more complex.

The historical trend of increasing cost with increasing complexity can be seen in Fig. 1, developed by Paul Eremenko at the Defense Advanced Research Projects Agency (DARPA). This historical schedule trends chart compares the linear growth of military aerospace development time with complexity with the flat trend of automobiles and the declining trend in the semiconductor industry. The inset box on the chart also notes the large exponential growth in development cost, doubling every 7 years in real dollars, compared to automotive development where real cost doubles every 20 years, and semiconductors, where real cost has not grown at all [1]. Something about the design of aerospace systems causes system development

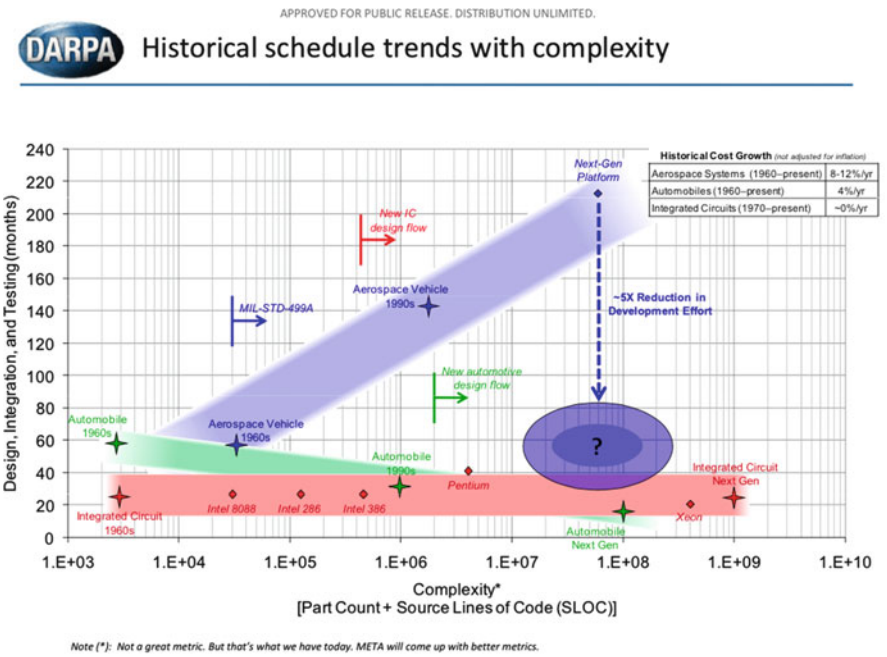


Fig. 1 Historical schedule trends with complexity for aerospace, automobiles and integrated circuits [1]

to become catastrophically more difficult (as reflected by growth in cost and development time) as the systems themselves become more complex.

The growth of development cost parallels the growth in production cost. Norm Augustine's 16th Law puts this very provocatively,

In the year 2054, the entire defense budget will purchase just one aircraft. This aircraft will have to be shared by the Air Force and the Navy 3.5 days per week, except for leap year, when it will be made available to the Marines for the extra day. [2]

This exponential cost growth, first observed in 1983 but updated in Fig. 2, is a phenomenon that has continued to the present.

Why did Augustine's graphs not have the ultimate effect of changing the status quo? Perhaps the answer lies in the complex nature of modern systems. The complexity slows down the project and increases the costs until it gets to a point that no amount of development time can pull it off. A good example of what can happen is the Army's Future Combat System (FCS), the largest and most ambitious planned acquisition program in the Army's history, which was cancelled early in development [3]. The system of systems approach to FCS included challenging software, integration, and life-cycle components [3].

FCS is not an outlier. In fact, the cancellation is an outcome we have come to expect. A value analysis of large Department of Defense development programs measured the combined impact of overruns, delays and cancellations as a loss of \$200 million per day on average over the past decade [4]. Accounting for complexity is a way to address these issues.

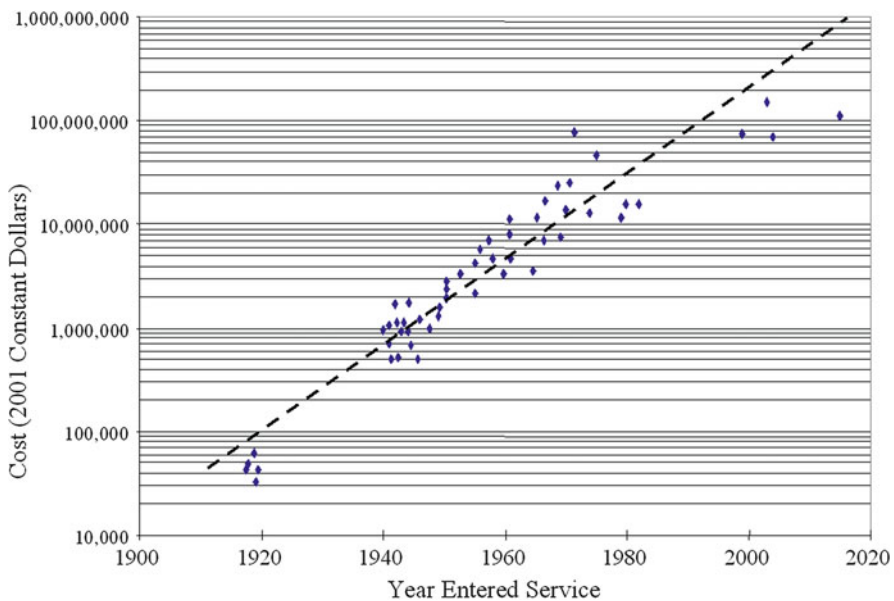


Fig. 2 Augustine's trend of fighter aircraft production cost versus time, updated with contemporary aircraft, and adjusted to use constant dollars. Augustine observed similar relationships for other military aircraft, ships and ground vehicles, and commercial aircraft

1.2 What to Do About Complexity

The view that systems engineering is very difficult on modern large systems because they are so complex seems reasonable. However, the practicing systems engineer needs more than a truism. He or she needs a pragmatic guide to assessing complexity, and based on that assessment, an appropriate approach to reduce, mitigate, adapt to, or (better yet) exploit complexity and bring large projects to successful conclusions. A complete roadmap from complexity to assessment to approach does not yet exist, but this chapter will discuss the beginnings of such a strategy. Note that some of this roadmap will lie outside of the realms of study normally considered to be the Systems Engineering Body of Knowledge. Consequently, the final product will certainly be transdisciplinary. Several of the methods discussed here are examples of that.

Discussions of complexity often revolve around emergent properties or behaviors of the system. Emergence is considered by many theorists to be the defining characteristic of complexity. There is considerable debate as to whether such emergent properties are objective or subjective. Are we surprised by system behaviors (and therefore label them as emergent) because we do not know the cause of the behaviors, or because the cause is somehow unknowable? This chapter takes the position that it does not matter. Whether system behavior is unpredictable because of ignorance or some deeper nature of the system does not matter—what does matter to the designer is the unpredictability itself, and the way unpredictability interferes with traditional system design and development.

2 Complexity

Before ways to deal with complex systems can be suggested, it is important to survey how complexity and complex systems have been defined and what techniques and methods have been tried to cope with it. The techniques and methods referenced here are not the result of rigorous analysis; they are heuristics developed over time, based on what seemed to be effective. In that regard these approaches tend to be non-hypothesis based methods.

2.1 A Survey of Complexity Definitions

Complexity lacks a single definition. In fact, it has a multitude of definitions. Seth Lloyd came up with forty or so [5]. Melanie Mitchell describes hosting a panel discussion between esteemed experts for the Santa Fe Institute's Complex Systems

Summer School in 2004. The first question to the panel was, “How do you define *complexity*?” None of the panelists’ definitions matched and some of them disagreed with each other [6]. The definitions that exist range from the highly formal to the informal, from the mathematical to the non-mathematical, from those based on trying to measure the amount of complexity to those who see it as unpredictable and, hence, unmeasurable. And yet, several categories of definitions exist.

Scott Page defines complexity as a collection of diverse, connected interdependent entities whose behavior is determined by rules, and which may adapt, but need not. The interactions of these entities often produce phenomena that are more than the sum of the parts. These are called emergent phenomena, and most complex systems are therefore not predictable [7].

One characteristic of complexity is that it lies Between Order And Randomness (hence, the BOAR complexity model), and another is that it cannot be easily Described, Evolved, Engineered or Predicted (the DEEP complexity model) [7].

Melanie Mitchell extends the definition beyond “complex” or “complexity”, suggesting that a complex system is one that exhibits nontrivial emergent and self-organizing behaviors [6].

A valuable adjunct to our definitions of complexity and complex systems is a description of the behaviors or attributes of complex systems since these are what we normally encounter when dealing with complexity. David Snowden and Mary Boone [8] describe complex systems as having the following characteristics:

- Large number of interacting elements
- Non-linear interactions such that seemingly minor changes can produce significant consequences
- Solutions emerge from dynamic circumstances, that is, they are emergent
- Elements of the system evolve together in irreversible ways
- Hindsight cannot lead to foresight because conditions and the system constantly change
- The system and the agents operating within it constantly operate on each other in ways that are unpredictable

Many of these attributes contribute to the unpredictable characteristics of complex systems. Large numbers of interacting elements means it is difficult or impossible to predict how a change in one element will impact another. And, the interactions can be highly non-linear; a small action may have a large effect and vice versa. These interactions vary with time so semi-stable states or solutions will often evolve and emerge over time in ways that could not be predicted in advance. And, once they have emerged, things will not go back to the starting position.

Paul Cilliers uses the wonderful example of making mayonnaise when he said, “a jumbo jet is complicated, but a mayonnaise is complex” [9]. That is, once built, each part of an airplane can be taken apart, analyzed, and then put back together. The result will still be an airplane. A person confronted with the constituent ingredients of mayonnaise would be hard pressed to predict what the result of

beating them all together would be and, once created, the mayonnaise cannot be “unassembled.”

Cilliers’ use of the words “complicated” to describe an airplane and “complex” to define mayonnaise also highlights one of the challenges of creating specific definitions for words that are in common English usage. Unfortunately, the words “complex,” “complicated,” and “chaotic” are often used as synonyms in common conversation. “It’s complex,” “It’s complicated,” and “It’s chaotic” can all describe the same situation. If you ask someone to define either complex or complicated you’ll find that it’s often difficult for them to define complex without using the word complicated and vice versa. This will be described more fully in the section covering the Cynefin Framework, but for our purposes, a complex system is inherently unpredictable while a complicated system could be predicted through analysis.

Complex systems are highly dependent on even tiny variations in starting conditions; consequently, you can never exactly replicate the same results. Consider, for example, a baseball game. The same two teams could play games on consecutive days and the games will not be identical even if it’s the same players on the same field in the same weather conditions using the same equipment and game strategies. It’s the inherent complexity that makes sports popular. If every game were completely predictable, sports would be dull.

2.2 *The Consequences of Complexity*

Part of the reason that so many definitions of complexity have been suggested is that there are many and varied aspects that appear to exist in complex systems. Consider, for example that complexity descriptions can include:

1. Technical complexity: Technologies can interact and integrate in ways that are difficult to determine in advance.
2. Business system complexity: Large organizations with multiple interrelated organizations or complex international supply chains can create complex relationships.
3. Managerial or organizational complexity: Similar to business systems complexity. A single entity developing a product with their own funds is significantly different from a project with a mix of private and government funding from multiple sources that may have different (and possibly competing) goals and objectives.
4. Scale complexity: Sometimes new ideas are piloted successfully in small experiments and yet fail when scaled up to a large organization or enterprise.
5. Coupling complexity: Describes the degree in which things are related to each other. The greater the number of interacting agents the more difficult it is to predict the results.

6. Interactive complexity: The amount of time or slack between interactions of elements in a system. A trigger mechanism for chaos is the inability to have time to reflect; the plan is moving forward all the time, people don't stop to look at what happened or go to another route.
7. Cognitive complexity: There are simply too many details and too many connections for any one person to fully understand the system.

People who must deal with complexity often focus their attention on one or more of these types of complexity. This can often be the result of the real or perceived causes of past problems. If a past program took on too much technical complexity ("invention on the critical path") for instance, a current program leader may focus significant amounts of attention on dealing with technical complexity...and completely miss a scaling or organizational problem.

Experience has taught people a series of rules of thumb, or heuristics, to try to avoid the problems often associated with complex systems development. Unfortunately, experience alone can be a notoriously poor teacher and may simply reinforce our own biases as to why things are happening and what to do about them [10]. Worse yet, our biases can decrease our ability to detect true root causes such that we continue to react inappropriately in our attempts to successfully manage complex systems.

Consider, for example, large projects of the type that are often government funded since they are so large as to all but require state-level funding. In his 2003 book *Megaprojects and Risk*, Bent Flyvbjerg describes this phenomenon by showing that huge infrastructure building projects are frequently not economically viable despite reams of initial analysis showing they were. The political nature of these infrastructure projects leads to exaggeration and distrust about the facts. Ultimately, the politicking threatens to bankrupt a company or even a country. Although projects suffer poor performance, the need for these infrastructure capabilities dictates that the projects not be canceled. Flyvbjerg suggests an approach specifically created to overcome the weaknesses of the conventional approach in megaprojects by emphasizing risk, institutional issues, and accountability [11].

Sometimes an appeal is made to decrease the complexity as a way to manage successfully. Consider an example frequently used by Dr. Eli Goldratt (creator of the Theory of Constraints). He drew the shapes shown in Fig. 3 and then asked the question, "Which is more complex?" [12].

Most people say that System B is more complex. Their sense of complexity has to do with the level of difficulty or the number of words needed to define the system (A version of Kolmogorov or Descriptive Complexity). Goldratt would say that, as a physicist, System A is more complex because it has many more degrees of freedom and it would be extremely difficult to predict how any input would influence the system since we have no information on how the circles are related. In contrast, we know exactly how any change in any element in System B would impact the rest of the system (as indicated by the arrows between the circles). This is why Goldratt maintained that any complex system must have a sense of "inherent

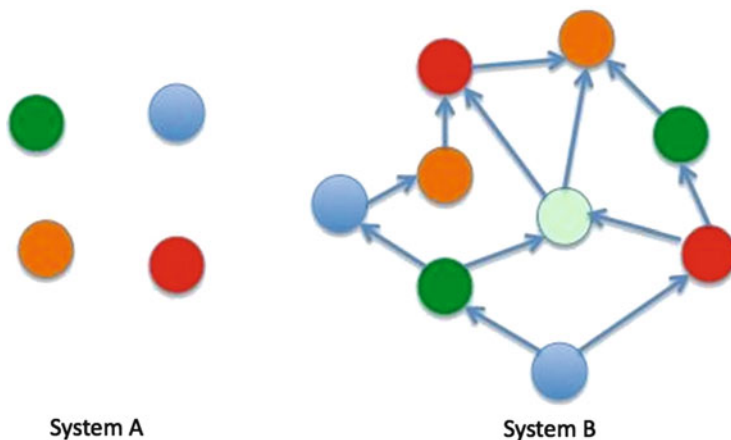


Fig. 3 Goldratt, creator of the Theory of Constraints, asks which is more complex, System A or System B? [12]

simplicity” behind it that would provide significant leverage for improvement. . .if we can find it.

A less mechanistic and more organizational approach is seen in the story of a software company in the US that set out to create a more empowered work force by flattening the organizational structure to essentially two levels: they had about a half dozen managers and 300 employees. The intent was that employees would interact with each other and self organize as required to accomplish tasks, solve problems, make improvements. What they discovered was that a new level of “phantom middle managers” began to appear. Initially these were simply people who had more knowledge about some aspect of the business than others and they gained a reputation for being helpful and informative. Essentially they became major nodes in the overall web of personal interaction. When this new phantom level began appearing, the actual management tried to break it up, hypothesizing that it would block the empowerment they were after. But they realized that this was a case where the right people, in the right positions, were being identified and organically promoted. It was essentially a successful, emergent management system that was not derived from any traditional means of designing organizations. And it fit the company and its culture.

Another way to think about adapting is through rapid, small scale testing of ideas. The approach described by Eric Ries in *The Lean Startup* [13] presents a new way of looking at the development of innovative products that emphasizes fast iteration and customer insight, a huge vision, and great ambition, all at the same time [13]. The approach can be applied both to startups and to internal teams of a parent company that act as startups. “If the fundamental goal of entrepreneurship is to engage in organization building under conditions of extreme uncertainty, its most vital function is learning” [13].

The lean startup method has five principles that accommodate the uncertainty found in complex systems:

- entrepreneurs are everywhere
- entrepreneurship is management
- validated learning happens through frequent experiments that test each element of the vision
- the experimentation is a build, measure, and learn activity
- innovation accounting, or holding innovators accountable

The Lean Startup approach is based on extensive use of experimentation to evolve products. But an organization following this approach does not do experiments just for the sake of experiments. The experiments are done by a small group for a short time and the results then inform the project. If the project team is smart about their experiments, they save months of ineffective time on assessment. It's not always known what is going to happen when the experiment is run, but the assumptions are set explicitly and then rigorously tested [13]. The intent is to maximize learning, not maximize failure as the key strategy. This sort of learning through tests and failures has often been interpreted as meaning that the goal is failures; it isn't. The goal is learning and often the most important lessons are with respect to what does not work.

3 Ideas to Build on

The issues raised in this chapter are not new, they have been puzzling the systems engineering and software engineering communities for decades. There is no final or complete answer, but a foundational approach is presented and some practical approaches to confront complexity in systems development can be useful.

Perhaps the most direct assault on the challenge of developing large, very complex systems is the Northrop Report from the Software Engineering Institute [14]. An important conclusion of Linda Northrop and her colleagues is that, when systems are very complicated, specifying requirements is no longer practical or even useful. Complex systems are defined by both the environment into which they will be inserted (or launched, or released), and the broad changes that are hoped for as the system interacts with its environment. What the system itself will be, what precisely is being built, is not knowable. It is not knowable ahead of time, and it is not precisely knowable even when the system is in the field. This unknowability is inherent when we say the system is "complex." What we need to do is face up to unknowability, and address the consequences.

To the extent that there are understood definite functions in such a system, they are conflicting and subject to rapid evolution. The components of complex systems are not a neatly ordered tree, they are heterogeneous, changeable and not-quite consistent. And, they will include fields of study beyond what is normally considered to be traditional Systems Engineering.

Felder and Collopy [15] identify more of the challenges facing systems developers today. The traditional systems engineering V or the waterfall process of functional decomposition, component design and test, and successive re-integration is impossible when systems have tens of thousands of components and the probability of any single component or interface working is less than 99 % [15].

We trusted systems in the past because we employed a strategy for verification testing that was ironclad:

- Test every system state
- Test every transition from one state into another
- Make sure each state and each transition behaves as intended

However, current systems, like the Airbus A350 or the F-35 fighter have a vast number of states and an astronomical number of state transitions. These cannot be tested. We cannot even test a representative sample of states and transitions, or test only the transitions that are most likely to occur. Instead, we are facing a future where the most likely type of system failure is a once-in-a-billion year occurrence that happens once and is never seen again. But there will be so many potential failures of this type that they can make systems utterly unreliable.

And yet, some large complex systems succeed. The landing of the Jet Propulsion Laboratory's (JPL's) rover Curiosity on Mars in August of 2012 was a triumph of modern engineering. JPL has accomplished several similarly stunning successes in the complex systems arena. Can we ask the Curiosity team how they did it, and follow their lead? Probably not. Donald Schön informs us that we will not find the answer by asking. When experts describe how they do what they do, they create an honest reconstruction of how they think their behavior must occur, but it is frequently very different than what they really do because expertise is generally unknowable to the expert [16]. Most expertise consists of tacit knowledge that is not expressed in words, concepts, or symbols. Much of what makes JPL successful is not describable by the people who work at JPL, and this means that their success cannot simply be transplanted to another organization.

One way to identify complex systems is by the nature of the developer. Individuals can develop simple and even complicated systems. A bicycle sits right on the boundary of what a person, working alone, can do well in development. If a good engineer devotes his or her career to understanding all the aspects of bicycle design, from metallurgy to tribology to dynamics to coatings, it is conceivable that they might be able to design and develop a very good bicycle all on their own.

For anything more complex than a bicycle, a team of designers is needed. No single person knows enough to design every minute part of an airplane. But the maximum effective team size is around 15 people.

Beyond 15 people, the team must be replaced by an organization, with roles and responsibilities, communication channels, and reporting relationships. This is the realm of complex systems development. Frederick Brooks managed one of the earliest very large development organizations, and wrote about it. He noted that communication presents a barrier to complexity, because once a design team

requires dozens of designers, they cannot all communicate with each other—there is not enough time in the day for all that communication to take place. Therefore, people need to be partitioned into groups, and communication among the groups needs to be channeled. The result is that important ideas cannot always be adequately shared, and the prevalent fault becomes “I thought the system was being designed to behave like x, but in fact it did y” [17].

Research suggests that the optimal size for a team is around seven people [18]. An extended group can operate effectively with minimal structure as long as everyone trusts everyone else. However, the maximum size of a group where such trust can practically exist is about 15 people. Team size also has an effect on the value of transparency. In a small team (15 or less) transparency of who is working on what and their progress is a benefit since it allows frequent integration. Essentially it is the design equivalent of small batches or Single Piece Flow. Learning from each other can be rapid. But as the team grows beyond that level there is increasing loss associated with transparency. Increasing numbers of people mean increasing coordination requirements, which nearly always reduces the time available to spend on the design. And, because the trust limits have been exceeded, it means that people will be in the position of exposing their “failed” designs to people they do not know and do not trust. This makes coordination more difficult and slows down organizational learning.

The work of Dr. Robin Dunbar suggests that there is another organization size breakpoint at about 150 people. This is the approximate number of people that we can be acquainted with at one time. That is, in an organization of 150 or less, each person would know each of the other people. This is a Skunk Works type development environment, and groups like this have been very effective, successfully developing systems such as the F-117 fighter and the SR-71 very high speed reconnaissance aircraft. However, a complete modern aircraft is beyond the reach of a 150 person team. Neither the F-117 nor the SR-71 was adequately designed for mass production or military maintenance and support, and that is why both are retired today. To design a F-35 requires tens of thousands of people, not just 150.

Still, large organizations can be very effective at developing simple or even complicated projects. McDonnell Douglas developed the F-18 E/F models on time and on budget, and the derivative aircraft has been very successful. Boeing and Airbus have become very effective at developing derivatives of their existing aircraft models such as the Boeing 767–400 (a fuselage stretch of the 767–300).

3.1 Process of Design and Design of Process

Product development is a process that is filled with uncertainty. Part of that uncertainty depends on the product being designed and part depends on the process being used. One person believes that the key to a successful design is a complete set of requirements based on an upfront understanding of customer requirements so the way to start is with requirements. Another is convinced that customers don’t really

know what they want until they see it, so the best approach is to start designing and then share the results with customers to get their feedback. Experiential and anecdotal evidence suggests that both work sometimes, but not always. There's no way to know in advance which is the best approach in a particular case. And, no matter which approach is used, it seems that things happen that were never in the plan but must, nonetheless, be dealt with. In this situation, what's needed is a more transdisciplinary "sensemaking" approach—a way to help guide us despite having incomplete information.

The process of design can be thought of as starting with a blank slate and coming up with a product. Analysis, on the other hand, is evaluating a pre-existing idea, concept, or design. This seems to imply that analysis must start with something and that design can start with nothing. The reality is a bit different in that a designer is the product of their own background and will quite naturally use their existing knowledge and experience to craft their designs. A designer, then will not truly start from nothing, they will start, even without realizing it, from a set of baseline requirements, many of which seem obvious when stated: An aircraft must have some way of moving through the air. It likely has some sort of propulsion system, some form of guidance and control system. If it has to carry a pilot then an additional set of requirements come up. If the project is to design a commercial jetliner, then it must carry passengers, it will have jet engines, it must have a way of fitting into the transportation network, etc. Realistically this means that many design details are given, no matter what sort of commercial jetliner it is. Designers know about how big humans are, they know things about jet engines, their fuel, how to control them, etc.

Then, if we consider characteristics of our desired airplane like payload and range or which airports it is intended for, we can use a series of analytical calculations to determine design parameters. What's left is the truly unknown parts; the "blank slate" elements of design.

If we step back and consider that process, that means that there are some aspects of design that are essentially given, other aspects that can be analytically calculated and finally, some that must be developed in a sort of trial and error, iterative approach. The given aspects are the simplest, the analyzed portions the next hardest and the "invented" parts the most difficult. In an actual design effort, we may be doing parts of each of these domains simultaneously. This can lead to confusion, design churn and rework, and conflict.

Two approaches to providing some guidance in this situation are presented, the Cynefin Framework and methodology associated with Agile Software Development.

3.2 The Cynefin Framework

The Cynefin Framework (created by David Snowden, Cynthia Kurtz and others) [20] provides a sensemaking approach to help identify the degree of complexity we

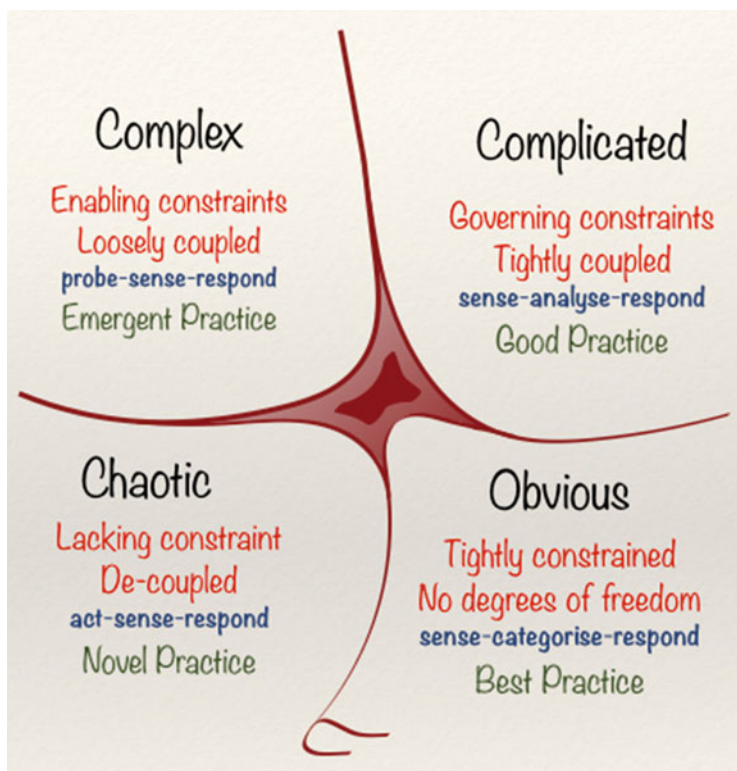


Fig. 4 The domains of the Cynefin Framework [20] (Image used with permission of David Snowden)

are operating in and determine an appropriate response. Knowing everything that is desirable to know is often not possible; sensemaking is a way to determine enough of what's going on to choose a reaction that matches the situation and increases the chance of success (or survival). As shown in Fig. 4, the Cynefin Framework includes a specific context for complex, complicated, obvious (simple) and chaotic systems.

The Cynefin framework is specifically illustrated with curved lines and no outer boundary to emphasize that it is not a 2 by 2 matrix and that the shape of the domains can change as our degree of understanding of the context changes. This is an important aspect of a sensemaking approach and is in contrast to a more deterministic categorization approach. An example of a categorization model is the 2 by 2 matrix used by Charles Perrow to illustrate Normal Accident Theory [19].

In general, the right side of the framework represents domains of Order in which cause and effect apply. The left side represents "unorder" in which cause-effect relationships are only evident in hindsight, if at all. The odd shape in the center represents Disorder, a state of not recognizing which domain you are in.

Obvious: The Obvious domain, previously referred to in literature as the Simple domain, describes a situation in which causality is clear and obvious to all observers. Consequently, the appropriate response is to follow a sense-categorize-respond approach. This is the legitimate domain of best practice [8].

Complicated: The Complicated domain applies where cause and effect still apply, but the relationship can only be determined by analysis or expert determination. Consequently the approach is to sense-analyze-respond. There is no longer a single best practice but there may well be several good practices [8].

Complex: The Complex domain describes a system in which a closed form solution is not possible. There are so many potential interactions in so many unknown ways that the observer cannot determine a “right” answer in advance. Consequently, the approach is to utilize probe-sense-respond. Interacting on the system with multiple experiments will lead to emergent practice. Note that after the fact, Complex domain interactions often appear as if they had been in the Complicated domain. This retrospective coherence, or 20/20 hindsight, frequently leads to the misperception that if only more analysis would have been done then the answer would have been determinable. This seems true, but it is not [8].

Chaotic: The Chaotic domain describes a system in which actions at the local level appear completely random. Not only is it not possible to analyze an approach (Complicated domain) but it is also not possible to develop coherent experiments (Complex domain). Consequently the appropriate approach is one of act-sense-respond which leads to novel practice. Novel practice sounds innovative and appealing until you realize that you can't plan on it ahead of time [8].

Disorder: The domain of Disorder describes the situation of not knowing which domain you are in. Within the Disorder domain, it is not uncommon for people to fall back on habitual behavior based on the domain they are most used to or most comfortable operating within. People who are experts in one problem solving method will, for example, tend to see all problems as solvable using their expertise.

The domain differences also suggest that the way that people operate most effectively within them should be different. And, as a project transitions from one domain to another the organizational structure should change to accommodate the new domain. The following are provided as general guidance for operating within the four main domains.

Obvious: For the cases in which there is one best way of doing things and everyone should be following it, the organization can have a strong, central authority overseeing the actions of hundreds or even thousands of people. Nearly everyone, for example, knows what a Stop sign looks like.

Complicated: In this domain experts and groups of experts must work together. This is where an organizational size limit of about 150 people makes sense. That allows them to all interact with each other while still having smaller groups of discipline experts.

Complex: In this domain the focus needs to be on relatively small, multi-disciplinary teams of people who trust each other, hence the limit of about 15. The group needs to be within the trust limits because emergence is expected and any of the team members might spot it first. Trust between team members is

beneficial to ensure that everyone acts in the best interest of the overall team. Note that the size of sports teams and military units are usually a bit less than 15. This is primarily an experiential heuristic, but seems remarkably consistent in human activities.

Chaotic: Because operating in the Chaotic domain requires speed and aligned action the teams should be small, 5–7 people. This is approximately the number of items we can hold in short term memory and appears to be correlated to the number of people that forms the most effective crisis management team.

As an example of how to use the Cynefin Framework in deciding how to react to a given situation, consider the two modes of learning described by James March. Low-Intellect Learning describes the situation where you copy what someone else does and if it works, you don't need to know the theory behind why it works [21]. High Intellect Learning is when you do understand the theory such that you can modify the practice to match your context. When you first encounter a problem, getting a sense of which of the domains of the Cynefin Framework it falls in increases your chance of being able to use High Intellect Learning and reduces your chance of relying on the often less successful approach of Low Intellect Learning.

The first goal of sensemaking is to establish a common basis of reference; to avoid scenarios like the blind men and the elephant in which each person may have an equally valid and defensible position. Although everyone's observations are correct, their conclusions are not correct. In a complex product development system each of these observers may be in a different discipline. Ultimately the only way to recognize the whole, which is the elephant, is by taking a transdisciplinary view of it.

Although the general construct is referred to as the Cynefin Framework, when applied to a specific situation it becomes a model of that situation. The application is the model. An example of building a model from the context of a specific situation is using a technique called Cynefin Contextualization or Four Corners. Basically, a group of issues, concerns or statements describe a particular situation. It works well to have each on a separate sticky note and then have a large piece of blank paper, a tabletop, a wall, etc. Note that the temptation is to draw the domains or domain boundaries in and then place notes within the boundaries. But if the domains are drawn first and then the issues are fit into them, this becomes a categorization approach and not a sensemaking approach. It's more informative to place the issues first and then draw lines later. Instead of drawing the domains, think of the four corners of the blank paper as the extremes of the four main domains. Place the issues by picking them up one at a time and imagine that each sticky note has four rubber bands attached to it and to each of the four corners. Using the four corner conditions as guides, place the sticky note where it best seems to fit. Some will be more toward one side or one corner. Others will tend to clump near the center. Be alert for differences of opinion among team members as to where a particular note goes. If, for instance, one believes it should go in the lower right and another in the upper left, create a second note and place both of them on the page. Come back to these notes later because the difference may be a sign that

people are looking at the same situation and seeing different situations. This is useful. Once all the issues have been placed, then the apparent domain boundaries can be drawn. Note that there may be more issues in one area than another, and there may be a number of issues in the center or Disorder domain, etc. Do not expect the result to be symmetric or the domains to be of uniform size. This context specific representation indicates how the domains are derived from the information (sense-making) as opposed to placing the issues into pre-selected domains (categorization). More detailed instructions can be found in Methods section of the Cognitive Edge website [22].

A related method is The Future, Backwards, which is akin to a cross between scenario planning and vision creation. It is primarily used to help learn how to recognize weak signals in a system. The process involves the creation of three simple state descriptions representing (1) the present, (2) a wonderfully positive future, and (3) a horribly negative future and then working backwards to create a stream of turning points. The turning point streams of the two potential futures intersect the turning point stream of the present in the past. This is a way of showing that all systems have already sown the seeds of their futures.

In practice, this method works well with multiple small teams working independently. It can be very effective if the members of each team are drawn from the same constituency (shared background, same organization, same role, etc.). This is ironic, since it is purposely NOT taking a transdisciplinary approach at this point as a way to call attention to the differences. Once the maps have been created, members of each group look in turn at the other team's maps to see what is the same, what is different and what is surprising about each of the other group's efforts. Frequently, people realize that the signals indicating whether an organization will tend toward a positive or a negative path are initially quite subtle. Also, it can be very informative to learn that one group's positive is another's negative [23].

3.3 Agile Development

As software became more of a part of daily life, software development programs grew increasing large and encountered complex system behavior. Due dates and budget commitments became increasingly more difficult to meet and user requirements were either difficult to determine or changed more quickly than code could be written. The Standish Group Chaos Report is often cited in industry to show that most software projects that are completed are over budget, late, and/or not what the users wanted [24]. The report also says that many projects are cancelled before completion. The initial reaction was to apply linear, deterministic systems engineering and more detailed, process-centric project management methods more rigorously: more tasks, more details, more milestones, more status reports. This led to the use of the Waterfall Method for planning and managing software development projects. Waterfall is a top down approach with a mandated series of development phases that are planned and scheduled in advance and worked

sequentially. Waterfall and similar methods were called “heavyweight” methods. Despite the effort and the intent of Waterfall, the increasing complexity of software did not result in appreciable improvements.

The explosive growth of software occurred at about the same time as the growing knowledge of complexity theory and software developers began looking to complexity for answers. This created a pendulum swing, of sorts, with the perception that software creation was always on the edge of chaos, that it was unplannable and had to evolve based on empirical evidence. Software development entered an experimentation phase and saw the development of a number of alternative development methods. Some of the ideas seem to be guided by a rejection of the heavyweight methods like Waterfall. This rejection led some to assert that software projects should not have any planning, no documentation, no process descriptions, no fixed budgets or schedules, and no management. In contrast to the heavyweight methods, the new methods were called Lightweight methods and included Scrum, Extreme Programming (XP), Feature Driven Development (FDD), etc.

This proliferation of Lightweight methods led to a meeting in February 2001 at Snowbird, Utah where a group of 17 proponents met to reconcile their different methods. They came up with the name Agile as the collective term for what they did and wrote the Agile Manifesto and accompanying principles to guide further development [24–27].

The Manifesto, or, more properly, the Manifesto for Agile Software Development, has since been translated into dozens of languages and signed by thousands of people. It is both an acknowledgement of past practice and an appeal for more flexible and responsive processes as a more effective approach for dealing with complexity. For example, one of the precepts is that the signatories value responding to change over following a plan. One of the key principles is that software that does what customers want should be created and delivered quickly and frequently. This allows for rapid learning and evolutionary development as well as the ability to respond to change rapidly. Given these principles, Agile methods usually have small teams that produce working software in frequent, incremental releases. They make use of very close coordination with customers and are open to changes in requirements even late in the development cycle (For the full text of the Manifesto and the 12 Principles of Agile Software, see: <http://agilemanifesto.org/>).

Agile has continued to evolve and draw in useful features from other disciplines. Notable examples include the inclusion of theory and applications from Lean and Theory of Constraints. Interestingly, because of the success Agile methods have seen at dealing the complexity of software development, these combined approaches are now being fed back into the development of non-software products where they are finding utility for their ability to deal with variation and uncertainty in the development process.

3.4 *Risk and Uncertainty*

We care about complexity because the outcome of complex system development projects is unpredictable. This is much like saying that the outcome is uncertain, but uncertainty is a broader and more subtle term than unpredictability.

When systems engineering projects develop technology, uncertainty is unavoidable. If the outcome of development were certain, there would be no need for engineering design. Manufacturing drawings could be prepared immediately. The result would be known at the start.

But when a development project pushes technology frontiers, engineering is necessary, and therefore the outcome is uncertain. Project planning estimates include many aspects of the outcome (schedule, cost, product performance, reliability, and so on), but the actual outcome may be better or worse than the estimate.

This presents a problem for systems engineering, because the basic processes of decomposition, construction and integration assume a deterministic world. The only area of systems engineering that moves beyond determinism is risk management. Can risk management handle the uncertainty inherent in systems development?

There are two problems with using risk management to cope with the uncertainty that is endemic in systems engineering and systems development. First, risk and uncertainty differ in important ways [28]. Risk addresses bad outcomes that might occur. Uncertainty encompasses all outcomes that differ from the expected outcome, some of which are bad and some of which are good. Engineers want to mitigate or eliminate risk. However, with uncertainty, the right approach, according to decision theory, is to maximize the mean utility of the outcome. Thus, better than expected outcomes impact the overall design positively. Second, the systems engineering risk management process assumes that there are a small number of risks to be addressed—at most, several dozen. However, in a high technology project, uncertainty is ubiquitous. For a new commercial airliner, there are tens of thousands of uncertain outcomes [29]. The design of every component and part has uncertain cost, mass, development schedule, and so on. Although the magnitude of most of these uncertainties may be small, the collective effect can have a huge impact on the success or failure of the overall project.

All these uncertainties must be managed. Also, uncertainty management must be an essential element of the foundation of systems engineering, not an add-on process, which is how risk management is currently structured.

As an example of the important difference between risk and uncertainty, consider 10 tasks that are sequential. The owner of each task estimates that their task is 90 % sure of success. From a risk assessment standpoint, each would rate their task as being of very low risk. But the overall project has a probability of success of 0.9 to the tenth power, or only about 35 %. That means that the uncertainty as to when the overall project will complete is very high even though the risk that any individual task will miss their due date is very low.

4 Common Threads

We need a new paradigm for systems engineering. Michael Griffin has suggested that Systems Engineering is a collection of heuristics that lacks a foundational theory [30]. The current systems engineering V doesn't work for complex systems. We need to overcome the perception that the reason things didn't work in the past was because we didn't try hard enough, this is frequently the results of retrospective coherence as described in Sect. 3.

We don't have it yet, but here are some elements toward a solution. The ideas that have been explored in this chapter reinforce the need for change and point toward a solution. We find that there are a number of useful principles that can be applied to complex systems and hope that a better defined process can be developed in the future. The better we understand the theory of what is happening, the better we can match practice with context.

4.1 *Implications of Complexity Definitions*

There are ways to be successful operating within complex systems, but you have to remain flexible, there are no recipes. Fred Brooks argues that you have to design the organization that designs the system. The design of the organization has to match the product being designed, even though it is uncomfortable, or at least unusual, for engineers to look at the engineering of organizations [31].

This understanding was slow in developing and still today many Systems Engineers see the role as primarily one of technical integration. Brooks's point is that Systems Engineering must look at product development as a whole system and the role must include the full socio-technical integration.

The opinion of attendees at a National Science Foundation workshop on the Science of Systems Engineering in November 2014 was that, in addition to the technical challenges to be faced, successfully navigating a complex development project will also require defining and/or accounting for mission context, markers for success, organizational and team culture; values, assumptions, and behaviors [32]. Since a number of those roles involve fields normally found outside of Systems Engineering, such as Sociology, Psychology, Organizational Development, and Behavioral Economics; a transdisciplinary approach is necessary in order to successfully integrate large scale, complex product development projects.

4.2 *Recommendations*

These recommendations represent some basic heuristics that will change for each instance in different ways. The common thread is to create an approach that does

not presuppose the solution in advance by employing agility, resiliency, flexibility, and constant feedback. Our recommendations can be summarized as:

- be able to react quickly to emergent risks and emergent opportunities
- use rapid, small scale, safe to fail experiments
- use small co-located teams of specialist generalists
- integration needs both lateral communication and vertical communication
- failure is a way of learning what works and what doesn't work

Being able to react quickly is one of the main attributes of Agile and is a part of the sensemaking aspect of probe-sense-respond advice from the Cynefin Framework. When applying a flexible process that accounts for emergent risks and opportunities, the normal bureaucracy will interpret the flexibility as neither ordered nor controlled. It will look like managers have no clue about how to manage, but it can't be presupposed that the system is deterministic. Elon Musk uses this approach very successfully with SpaceX. Up until now, we have been convinced at some level that the appropriate response to complexity is to impose more order on the systems engineering approach, not less. Recall from Sect. 1 how the Future Combat Systems was a failed attempt at trying to impose more order by insisting on repeated use of unrealistic timelines.

Use of rapid, small scale experiments enables insights through learning and/or failing experiences. The experiments are a way to accommodate the unpredictable aspects of the project. In some cases you need to persevere, and in some cases you need to pivot your approach to a new direction. If the experiments are truly safe-to-fail, they have the potential to spur innovation. When Nick Swinmurn wanted to test his hypothesis that people would buy shoes online, he worked with local shoe stores. He took pictures of the shoes, posted the pictures online, and when someone purchased the shoe, he bought the shoe from the store and mailed it to the customer. Yes, people will buy shoes online and that's how Zappos got started. At that time, it was not a valid business model, but his simple experiment proved that the concept was viable [33].

The more speculative the technology is, the more complex it is, the more need exists for a small team of specialist generalists that are co-located. The co-location is desirable in order to encourage rapid, frequent communication. If what is being built is a repetitive, mature product, it is easier to use a spread out organization, which is not as dependent on the need for frequent communication. Co-location works on a small scale but not on a large scale.

Vertical integration is important for new technologies because meeting customer requirements is very challenging without a carefully tailored product design. The first computers had to be built with all specially made parts, but this is not the case anymore. Now individual modules are available that just snap together and they work. Computers now use diverse supply chains, and it is possible to mix and match components. Fifty years ago, it would have taken a small team of experts to design and integrate all the systems in a computer. Now, modularity has reduced the need for that same level of lateral communication for systems integration [34].

Last, but not least, is that failure is a very useful experience. We don't learn from failing all the time, we don't learn from succeeding all the time either. Through failure, the limits of the solution are revealed.

5 Summary

A new paradigm is needed for systems engineering to account for the unpredictable nature of complexity. Standard systems engineering techniques used to create large modern products have resulted in an exponential growth in time and development cost due to complexity. Simply adding layers of modification to the existing paradigm will not work because complex problems today cannot be solved using deterministic or stochastic methods. As evidenced by the growth of integrated hardware and software for large aerospace systems, standard systems engineering is not obsolete, but it will not work for everything anymore. Of course not all systems are complex, and it is still very effective to use a closed form systems engineering solution for deterministic systems. If the final design can be accurately predicted in the early design phase, it is not a complex situation in need of an innovative solution.

When complexity is encountered, it does not need to be eradicated—it can be managed. Definitions of complexity vary, but most include emergent behavior in the system. Complex systems have a large number of interacting elements where minor design changes can have major consequences. And a solution that works in one context does not necessarily work in another. The inquiry is the same; the outcome is not [8]. Also, accounting for complexity is not just about finding a technical solution for an unpredictable system. Complexity is found in every facet, such as business, organizational, scale, coupling, interactive, and cognitive aspects of the system.

The Cynefin Framework can be employed as a sensemaking vehicle to interpret the character of the system by revealing what is complex, complicated, chaotic or obvious. Cynefin is a meta-strategy that opens a window to transdisciplinary thinking as a theoretical basis for the process of design that includes the design of the process both organizationally and technically. Agile represents an existing creative approach developed for software, which is now being introduced into hardware applications. Agility is the ability to both create and respond to change in order to profit in a turbulent business environment [35].

Cynefin works to illuminate a clearer picture of the context. Agile methods are an example of appropriate tools that fit in the appropriate context, but lack a fundamental theory to explain why they work [36]. Creating a foundational theory for both of these approaches will lead to the ability to manage effectively in complex systems.

For now, the best approach is to expect the unexpected such that the team is able to react quickly to both emergent risks and opportunities. Rapid feedback using small scale experimentation and finding out what doesn't work has also been found

to be extremely useful. A small co-located team of people with diverse background works well along with effective lateral and vertical communication within the organization. For complex system development, it is best to accept that uncertainty is inherent and unavoidable, there are ways in which it can be accommodated, and it can provide significant opportunities to those who know how to take advantage of it. And that is why all engineers should care about complexity.

References

1. Eremenko, P. (2010, October). *Adaptive Vehicle Make (AVM)*. Proposers' Day Briefing. Tactical Technology Office, DARPA.
2. Augustine, N. R. (1983). *Augustine's laws* (2nd ed.). New York: American Institute of Aeronautics and Astronautics.
3. Pemin, C., Axelband, E., Drezner, J., Dille, B., Gordon, J., Held, B., et al. (2012). *Lessons from the army's future combat systems program*. Santa Monica, CA: RAND Corporation.
4. Maddox, I. D., Collopy, P. D., & Farrington, P. A. (2013). Value-based assessment of DoD acquisition programs. *Procedia Computer Science*, 16, 1161–1169.
5. Lloyd, S. (2001, August). Measures of complexity—A nonexhaustive list. *IEEE Control Systems Magazine*.
6. Mitchell, M. (2009). *Complexity: A guided tour*. Oxford: Oxford University Press.
7. Page, S. E. (2011). *Diversity and complexity*. Princeton, NJ: Princeton University Press.
8. Boone, M., & Snowden, D. (2007, November). *A leader's framework for decision making*. *Harvard Business Review*.
9. Cilliers, P. (1998). *Complexity and postmodernism: Understanding complex systems*. London: Routledge.
10. Kahneman, D. (2011). *Thinking, fast and slow*. New York: Farrar, Straus and Giroux.
11. Flyvbjerg, B. (2003). *Megaprojects and risk, an anatomy of ambition*. Cambridge: Cambridge University Press.
12. Goldratt, E. (2008). *The choice*. Great Barrington, MA: North River Press.
13. Ries, E. (2011). *The lean startup: How today's entrepreneurs use continuous innovation to create radically successful businesses*. New York: Crown Business.
14. Northrop, L., Feiler, P., Gabriel, R. P., Goodenough, J., Linger, R., Longstaff, T., et al. (2006). *Ultra-large-scale systems: The software challenge of the future*. Pittsburgh, PA: Carnegie Mellon University, Software Engineering Institute.
15. Felder, W. N., & Collopy, P. (2012). The elephant in the mist: What we don't know about the design, development, test and management of complex systems. *Journal of Aerospace Operations*, 1, 317–327.
16. Schön, D. A. (1983). *The reflective practitioner: How professionals think in action*. New York: Basic Books.
17. Brooks, F. P., Jr. (1982). *The mythical man-month, essays on software engineering*. Reading, MA: Addison-Wesley [Originally published in 1972].
18. Levitan, B., Lobo, J., Kauffman, S., & Schuler, R. (1999). *Optimal organization size in a stochastic environment with externalities*. Santa Fe Institute Working Paper: 1999-04-024. <http://www.santafe.edu/research/working-papers/abstract/c3b337d659e39e8ef7e6e02c1f21f292>
19. Perrow, C. (1999). *Normal accidents: Living with high-risk technologies*. Princeton, NJ: Princeton University Press.
20. Snowden, D. *Cynefin framework image*. http://commons.wikimedia.org/wiki/File:Cynefin_as_of_1st_June_2014.png
21. March, J. G. (2010). *The ambiguities of experience*. Ithaca, NY: Cornell University Press.

22. Cognitive Edge website. Ref: <http://cognitive-edge.com/library/methods/four-tables-contextualisation-basic/>
23. <http://cognitive-edge.com/library/methods/the-future-backwards-basic/>
24. Hibbs, C., Jewett, S., & Sullivan, M. (2009). *The art of lean software development*. Sebastopol, CA: O'Reilly Media.
25. Shalloway, A., Beaver, G., & Trott, J. R. (2010). *Lean-agile software development*. Upper Saddle River, NJ: Addison-Wesley.
26. Anderson, D. J. (2004). *Agile management for software engineering: Applying the theory of constraints for business results*. Upper Saddle River, NJ: Prentice Hall.
27. Highsmith, J. (2004). *Agile project management*. Boston: Addison-Wesley.
28. Reinertsen, D. G. (2009). *The principles of product development flow: Second generation lean product development*. Redondo Beach, CA: Celeritas Publishing.
29. Collopy, P. D. (2015, March). *Technical risk management*. *IEEE Aerospace Conference, Big Sky, MT*.
30. Griffin, M. D. (2010). *How do we fix systems engineering? (IAC-10.D1.5.4)*. Prague: International Astronautical Congress.
31. Brooks, F. (2010). *The design of design: Essays from a computer scientist* (1st ed.). Reading, MA: Addison-Wesley Professional.
32. Collopy, P., & Mesmer, B. (2014, November). *Report on the science of systems engineering workshop*. AIAA 2015-1865.
33. Donelan, J. (n.d.). *Do lean startup principles have a place in the enterprise?* <http://thenextweb.com/entrepreneur/2013/08/06/do-lean-startup-principles-have-a-place-in-the-enterprise/>
34. Christensen, C. M., Raynor, M., & Verlinde, M. (2001). Skate to where the money will be. *Harvard Business Review*, 79(10).
35. Highsmith, J. (2002). *Agile software development ecosystems*. Boston: Addison Wesley.
36. Snowden, D. (2015, July). *Cognitive edge, of practical wisdom*. <http://cognitive-edge.com/blog/of-practical-wisdom/>

Transdisciplinary Perspectives on Complex Systems

New Findings and Approaches

Kahlen, F.-J.; Flumerfelt, S.; Alves, A. (Eds.)

2017, X, 327 p. 89 illus., 55 illus. in color., Hardcover

ISBN: 978-3-319-38754-3