Welcoming Software Into the Industrial Fold

James M. Sutton
Lockheed Martin Aeronautics

Software has long been the odd man out in business: It operates in ways that are different than, and often incompatible with, the disciplines of other industries, even when it is teamed with those disciplines under the same enterprise. It generally underperforms other industries on productivity improvement, integration success, quality, and customer satisfaction. Applying Lean production to software enables it to become an industry that works well with classic industries. Lean greatly increases software's contribution to enterprise and customer success.

Software development can greatly improve its business performance by discovering and embracing its kinship to classic (non-software) industries. Perhaps the most important thing software has to gain is guidance on how to implement Lean production.

Lean production as we know it today began in the automotive industry, first with Ford in the early 1900s, then evolving rapidly with Toyota in the 1950s and beyond. Lean is, at heart, a model for maximizing productivity. It turns the assumptions of mass production – the previous model for productivity – on its head. Over the last 50 years, nearly all the classic industries have moved to the Lean model. These industries have, on average, doubled their productivity while tripling their quality [1]. They have also improved their ability to integrate components into systems (integratability) to please their customers.

Software evolved separately from the classic industries and never adopted Lean[1]. Could Lean production, as it is understood and applied in the classic industries, improve software development? First, we show evidence that Lean not only works for software, in some ways it works even better than it does for the classic industries. Then we give an overview of one incarnation of a Lean software process.

What Can Lean Do for Software?

In a nutshell, Lean is about maximizing value and minimizing waste. This means that Lean projects do what matters to business success, and *only* what matters.

Software grapples with value and waste like any other industry. Several software programs at the author's company, Lockheed Martin Aeronautics, have used Lean techniques similar to those practiced in classic industry. The following sections discuss the effects Lean has upon each of the success areas mentioned previously: productivity, integratability, quality, and customer satisfaction.

Productivity

The size of software systems on aerospace programs has grown more than 100 fold since the mid-60s. Early systems like the C-5A weighed in at 50 thousand software lines of code (SLOC) or less. Current systems such as the F-35 are projected to reach six million SLOC or more at delivery. Other domains have grown at least as much, challenging the ability of software development approaches to keep pace.

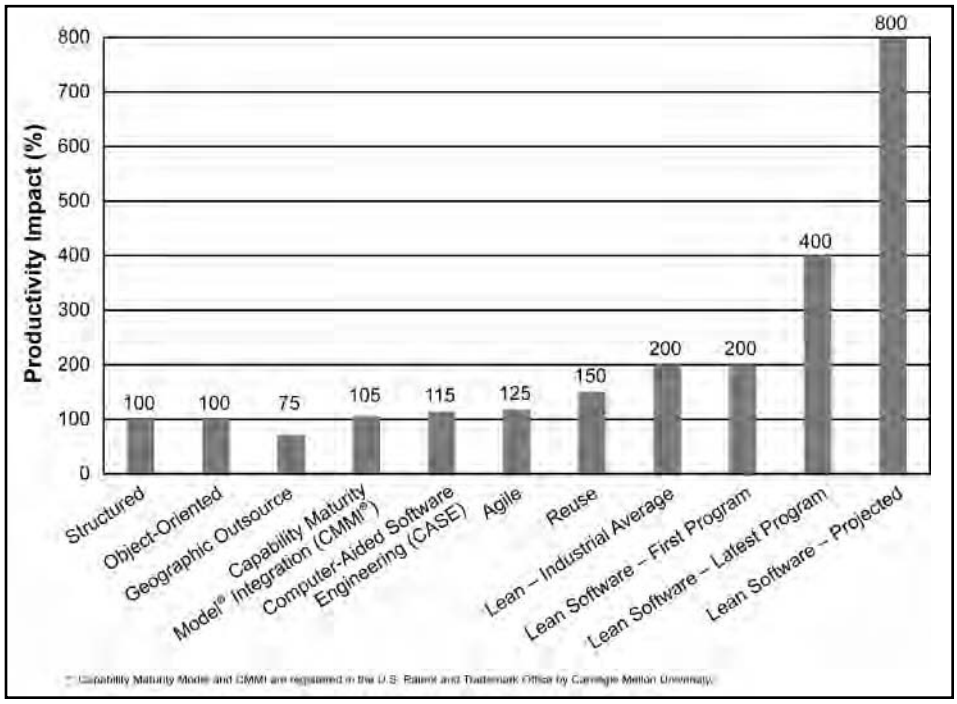
How much has the productivity of traditional software development improved since the early days? The left side of Figure 1 shows results published by proponents [2, 3, 4, 5, 6, 7] of various software approaches, with the structured techniques of the 1970s taken as the beginning reference value. Published claims are accepted without critical scrutiny but are averaged where multiple figures are cited[2]. All these numbers have been rounded up to the nearest multiple of five.

Even the 50 percent productivity gain from reuse, the most impactful non-Lean method listed, does not come close to keeping up with the 10,000 percent plus growth in software system size over the years.

Now to Lean. The industrial average figure for Lean comes from the classic automotive industries via the International Motor Vehicle Project (IMVP) [1]. The Lean software figures come from Lockheed Martin programs that have applied Lean principles to software. Those metrics have been collected and vetted through standard company processes.

The first reasonably full application of Lean to software in the author's experience occurred on the 382J program, a commercial upgrade of the C-130 airlifter [8]. Its productivity improvement matched the average improvement for classic industry due to Lean. Of course, individual businesses in classic industry often exceed a 100 percent average increase. Later Lean software projects exceeded this as well, ending at 400 percent on the latest program, the C-27J. The projected num-

Figure 1: *Productivity Improvements*



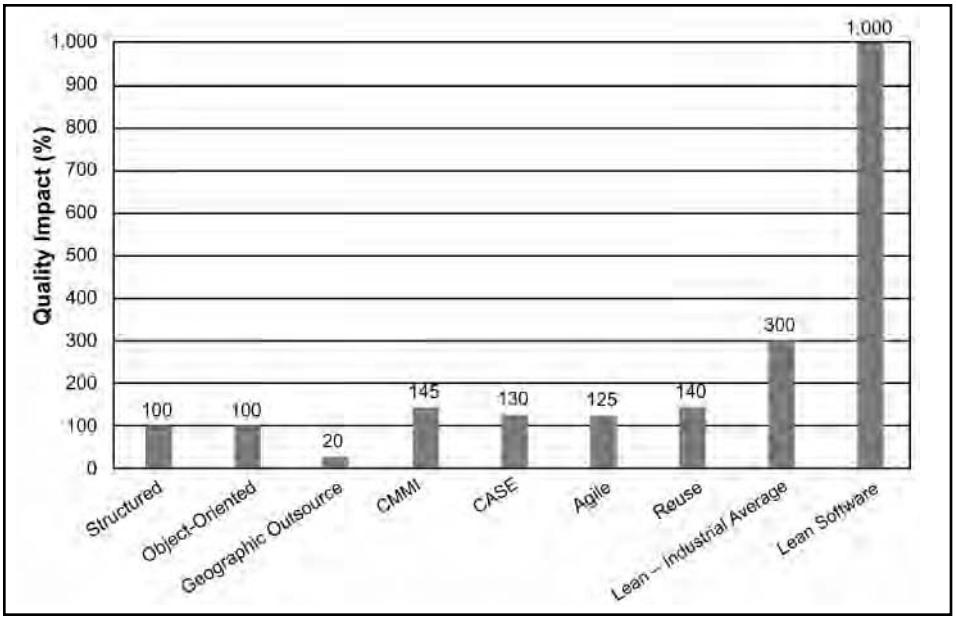


Figure 2: *Quality Improvements*

ber is based on a straightforward extension of process and tools identified by the software team.

Software productivity responds well to application of the Lean principles. Nevertheless, even 800 percent productivity growth is not enough on its own to make 10,000 percent-plus bigger programs successful. For that, major improvements must also be made to integratability.

Integratability

Source-line productivity can be increased by making a concerted push to speed up coding, but any gains are generally at the expense of the build process. In these conditions, errors are more likely to be introduced and less likely to be detected. Downstream builds are less likely to work correctly on the first try. Big projects require that the software coming out of integration be correct both functionally and structurally each time, and be on time. They can neither afford a process full of ongoing repair and reintegration loopbacks nor the cascading delays they bring to the rest of the project.

Programming languages and design structure set the ceiling for software integratability. Most of the widely accepted software languages are better at facilitating rapid source-line production than successful integration. As professor and head of the computer science department at the University of Northern Iowa, Dr. John McCormick recorded what happened when his students formed teams to program and integrate small systems of around 15,000 SLOC:

> During the first six years … students developed their control code in C. No team successfully implemented minimum project requirements. To ease student and teacher frustrations I made an increasing amount of my solutions available to the teams. Even when I provided nearly 60 percent of the project code, no team was successful in implementing the minimum requirements. [9]

The process breakdown occurred in integration. Remarkably similar results have been observed for C's successor, C++. Moving to a language optimized for integration (Ada) finally solved the problem for McCormick's students. It also worked for the Lean software projects mentioned previously (SPARK and Ada). But resistance to such change remains high among software professionals, and with good reason. Rewards and recognition are given for increased productivity in SLOCs (and, occasionally, quality), but almost never for builds that reliably work the first time as expected. Programmers are not to blame; the problem lies with rewards in the current software culture.

Classic industry has been able to use Lean techniques to reduce production cycle times – including integration – by up to 90 percent[3]. Adapting such techniques to the Lean software projects allowed *Lego-Block-like* assembly. Builds were developed and released once per week for several years. Each cycle began with new or modified requirements (an average of five, many of them complex) and ended with verifying that the entire system was still sound. This eliminated the typical *fix 1 SLOC*, *add 1.4 SLOCs of error* experience. With only a handful of exceptions, builds always worked correctly the first time, even though one of the programs, for reasons outside the software process, needed major releases four times as often as usual.

By facilitating significant growth in integratability and productivity, Lean showed a way for software to keep up with the exponential growth in system size experienced to date. Whether software will be able to keep up with future exponential growth remains to be seen, but the Lean projects to date have not even implemented everything in the Lean principles yet.

Quality

Quality, as we use the term, relates to code and supporting-artifact defect densities. Approaches that improve productivity often improve quality even more, as seen in Figure 2.

As before, all numbers for the non-Lean approaches are obtained from their proponents[4] and are reported as published[5].

The Lean software number comes from customer-sponsored independent verification and validation that compared the complete delivered Lean product to the delivered software systems of other vendors on the same aircraft. The results exceed those of classic industry by a considerable margin. It is probably not because this is an exceptional application of Lean: it is, after all, still an early experiment. A more likely explanation is that software is a nearly ideal candidate for Lean. It lacks many of the limitations that apply to material goods (e.g., inherent error tolerances) or services (e.g., soft or human psychological issues).

A metric that Lockheed Martin Aeronautics has used to keep things real is productivity times quality (divided by 100 to retain a percentile scale). One can easily improve productivity at the expense of lowered quality, or improve quality at the expense of lowered productivity. The multiple gives a better idea of how much help Lean – or any other approach for that matter – is actually giving to the software development process. On that measure, Figure 3 shows the software improvement approaches giving the best results, rounded up as before, and in order of increasing effectiveness.

This measure shows that Lean is not only effective for software development; its effects are balanced across both productivity and quality.

Customer Satisfaction

Few metrics have been kept of customer satisfaction in the software field. Evidence

is largely anecdotal, difficult to compare between competing approaches, and subject to challenge. Nevertheless, the Lean software efforts referenced in this article have their share of customer stories.

At the final acceptance board meeting for the 382J software, the United Kingdom acceptance authority said "This is the most nearly perfect software process we've yet seen." The same authority had previously refused to flight-certify several helicopters, which had sat inactive on a tarmac for more than a year because they lacked sufficient evidences of integrity to satisfy his standards.

An early attempt at applying Lean ideas (pre-382J) delivered software to a very picky department head at Tinker Air Force Base in Oklahoma City. This customer was known for returning everything to vendors for rework. The system, called the C-5B Ground-Processing System Software Update Program, was used for five years and no bug was ever reported back to the developers. The department head dubbed the development team "the new breed."

The customers of all the Lean software projects have made positive comments about the software they received. None has given any negative feedback.

A Lean Software Process

As noted earlier, Lean production starts with principles and culture. Lean culture is based on human respect, as well as a general expectation that people will spend most of their time doing things that actively benefit the enterprise or the customer (i.e., adding value). One survey [10] estimated that Toyota employees spend 80 percent of their time adding value, while employees of non-Lean businesses spend just 20 percent. Software is no exception, and there is a lot of room for improvement.

The Lean principles are as follows [11]:

- Value: Being enterprise- and customer-driven.
- Value stream: Preventing erosion of value and eliminating waste throughout the life cycle.
- Flow: Removing discontinuities between development tasks.
- Pull: Starting process control with the customer and passing it *upstream*.
- Perfection: Making it impossible to introduce defects, or else catching them quickly.

The means available for implementing these Lean principles include processes, methods, techniques, and tools. Nevertheless, Lean must actively address culture and management philosophy while

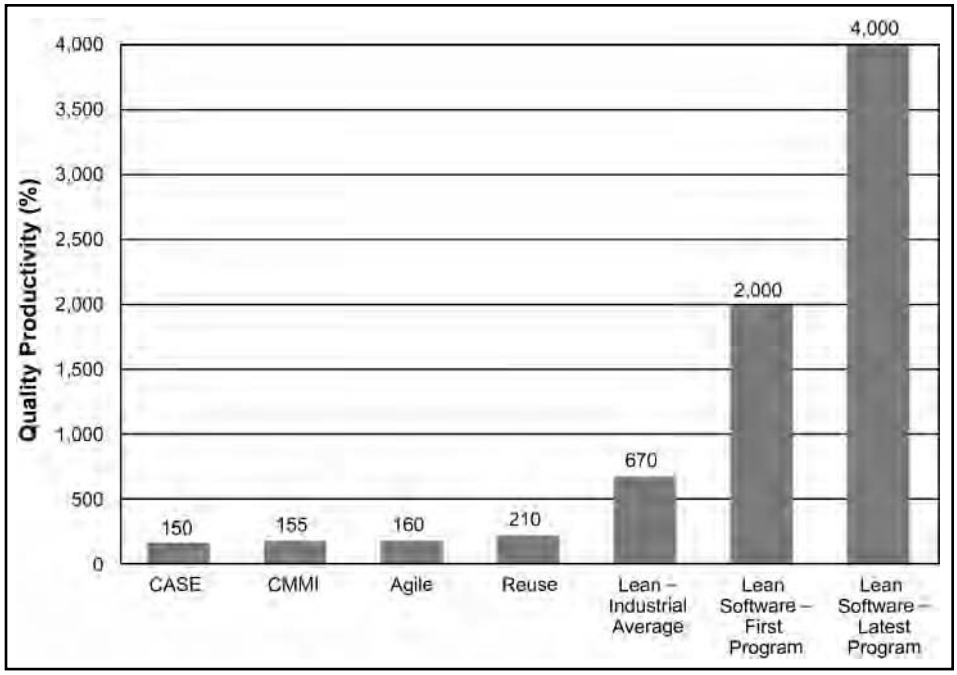


Figure 3: *Productivity Times Quality Improvements*

– and preferably before – it reworks processes, methods, techniques and tools. Otherwise, the Lean improvements will never fulfill their potential and what is gained will erode over time. However, that is a separate topic.

How the Lean principles are implemented in software is discussed in the following section, under the major software lifecycle activities of requirements, design, coding, integration, and verification. For brevity's sake, the techniques cannot be fully explained; however, references are provided.

Requirements

After developing a solid business case and obtaining sufficient project resources, the next most important success factor for software projects is to develop the best possible understanding of what the customer wants. In traditional software development this understanding comes in the form of upfront requirements. Agile projects converge upon it gradually through iterations of product releases and ongoing customer feedback. Both approaches have pluses and minuses.

However, what both traditional and Agile often miss is the customer's motivations. Lean calls these *values*. They include needs, wants, preferences, and responses to problems. Some values may not directly relate to specific aspects of the product at hand, yet could influence what goes into the product. For instance, different cultures process written information in different ways (e.g., right to left, bottom to top, and the significance of different colors). If you target a product to a culture other than your own, your user interface may be workable but not *just right*. They might not even be able to verbalize this; they will just know that they prefer the product of a competitor who understands them better. Identifying these things can be called *value resolution*. It is the foundation of a Lean product life cycle.

Understanding customer motivations will often lead you to requirements the customer might never have come up with on their own; ideas that not even an Agile team would have identified. Thus, value resolution and requirements analysis are different yet complementary as shown in Table 1.

Methods for identifying values include canvassing (literature searches, market

Table 1: *Value Resolution versus Typical Requirements Analysis*

	Value Resolution	Requirements Analysis
Focus	Customer Priorities	Product functionality
Expansiveness	Product line/Future oriented	Single product/Present or past oriented
Main Concerns	Customer delight, business	Organizational "realities"
Detail Level	"What" and "When"	"What," "When," "How"
Other Concerns	Systematic	Sporadic
Customer Involvement	Central	Peripheral

surveys, focus groups), brainstorming, the five why's (asking *why* something is the way it is, then *why* to the answer, and so forth until reaching the root cause), techniques from the Harvard Negotiation Project [12], domain analysis (identifying common and recurring characteristics of the customer's world), and Affinity Diagramming (one of the seven Management and Planning tools that involves grouping lower-level desires or preferences until higher-level values have been determined) [13].

Once you know the customer's values, the next step is to prioritize and, ideally, characterize them. Prioritization can be done with the Analytical Hierarchy Process, a technique for prioritizing a list by pairing all the items in the list, comparing the pairs, and using matrix math to generate a high quality quantitative ranking for each item in the list. This process has been used extensively in must-do situations such as international arms control [14]. Characterization can be done using Kano Modeling, a technique originally developed for the consumer photography market, to identify how customers will react to various potential product characteristics; either *must have*, *delight*, or *proportional* to how well they are done. Kano modeling can also be used on values and requirements [15]. The information these two techniques provide helps the program keep the customer *sold* and engaged, and uses program resources more efficiently.

Requirements are developed from values, and create specific guidance for the particular project. Table 1 gives some idea of the types of information to add (however, minimize negatives like *peripheral customer involvement*). For instance, textbooks state that requirements must omit *how* information. In reality, software development is almost always embedded in a larger development process where requirements must account for constraining higher-level system architectural decisions. There is almost always a certain amount of *how* either stated or assumed in real-world development. Those must be omitted from values, but must be adequately considered in requirements.

Leveraging the Lean Flow and Perfection principles depends on having requirements that possess four important qualities: completeness, correctness, non-redundancy, and unambiguity. The combination of these is sometimes called *requirements integrity*. It minimizes rework backflow (aiding Flow) and prevents many types of errors from entering the process in the first place (implementing Perfection). Two good approaches for achieving these qualities are the Four-Variable Model (FVM), a way of modeling the desired effects of a system on the characteristics of its surrounding environment as functions of the current state of characteristics of that environment [16], and Software Cost Reduction (SCR), a method that applies FVM to the specification of software systems [17]. SCR-style requirements are particularly well suited for embedded systems [18].

Design

Design defines product structures that are optimized for the customer's usage environment. Optimal structure ensures the system will achieve its purposes. It also minimizes the work involved in translating the functionality represented by the requirements into the form needed to go into that structure for implementation. The ideal is one requirement in just one place in the structure (e.g., one method or module in one object). That minimizes the non-value added work done between the productive work of requirement and design. This is the primary goal of Lean Flow. The Lean software projects cited earlier achieved this by developing an architecture that matches the structure of the FVM itself; then, SCR requirements had a natural place to go. This strategy also serves Lean Pull as it becomes easy to add functionality just in time; for instance, as needed to get particular features to the customer in interim releases.

Modern Quality Function Deployment (QFD) [19] is a method that assists with identifying the work that is needed to add value, and provides the means to track that work to assure everything gets done. It typically takes just days to perform and, in return, cuts weeks or months of waste from the software process.

Coding

The Perfection principle says to eliminate the causes of defects before the defects themselves can be introduced. Some programming languages do this better than others. SPARK eliminates all language ambiguities and is optimized for interfacing and for static (i.e., without executing) analysis. Ada has some of these characteristics, and has more options. The Lean software projects used both.

Problematic aspects of C and C++ from a Lean perspective were noted earlier. Their use does not completely negate Lean, but it does leave a lot of unnecessary and removable waste in the development process.

Integration

The Lean Principle most applicable to integration is Flow. Flow removes discontinuities between production steps. Integration seeks to make the code pieces combine immediately and effortlessly, allowing quick progression into verification and delivery.

The most useful Flow technique for software has been Design for Manufacture and Assembly (DFMA). DFMA leads to designing components that are *big parts*, i.e., parts that reduce the total component count but also simplify their integration. In software object-oriented (OO) terms, this means fewer, but generally not bigger classes, covering more of the total system. Standard OO alone will not lead to *big parts*; that requires a suitable abstraction of what software is and does. The concepts involved are mostly alien to software literature and are rather extensive, though not overly complicated so are explained elsewhere [8]. Dr. David Parnas' work was especially useful in developing a software DFMA approach.

Verification

Effective verification is the natural result of applying Lean to all the other activities. Spreading the implementation of requirements across various parts of an architecture, as is typically done in traditional design, forces the verification effort to correctly identify the extent of that spreading and also account for side-effects due to interactions between the pieces and other requirements found in the same design areas. All this work is unnecessary and can be eliminated if requirements have integrity and are isolated within the design.

SCR-style requirements remove the need to develop separate test cases for requirements-based testing. Such requirements are fully suitable test cases in their own right. This eliminates another development effort that is substantial on many programs.

A programming language that permits no ambiguities reduces the need for special verification techniques to detect compiler-based differences and unaccounted for paths (e.g., Federal Aviation Administration, modified condition/decision coverage testing). Such tests may still have to be performed for regulatory or legal purposes, but based on experience with the Lean programs they

will go quickly and uncover few defects.

Conclusion

Lean production has, on average, doubled productivity and tripled quality for the classic industries. Early applications of Lean to software have exceeded those results. Software development is an ideal subject for Lean because its product is pure information that lacks the physical limitations of durable goods as well as most of the soft issues of service activities. In software development, Lean can remain focused on the primary issues of value and waste. This allows the Lean tools to work with unusual effectiveness.

One of the biggest challenges for software development approaches has been to keep up with the growth in size and rigor of customer systems. Lean scales up easily for large systems. It works well in plan-ahead life cycles such as the Department of Defense acquisition system. Lean provides the evidences and assurances needed for safety-critical and high-security applications. These capabilities make it well-suited for the defense and aerospace domains, and for most other domains as well.

It is time for software to take its place as a classic industry and leverage the strengths of Lean production. Lean enables faster code production, smoother integration with other products, fewer surprises to budget and schedule, better quality, and happier customers. Lean converts software from management's biggest worry into one of its best means for assuring business success. Embracing Lean ushers software into the fold of classic industry as a welcome and synergistic partner.◆

References

1. Womack, James P., Daniel T. Jones, and Daniel Roos. The Machine That Changed the World. New York: Free Press, 2007.
2. Potok, Thomas E., Mladen A. Vouk, and Andy Rindos. "Productivity Analysis of Object-Oriented Software Developed in a Commercial Environment." Software – Practice and Experience. 1999 Vol. 29, No. 10: 833-847.
3. Ramasubbu, Narayan, and R.K. Balan. Globally Distributed Software Development Project Performance: An Empirical Analysis. Proc. of the 6th Joint Meeting of the European Software Engineering Conference and the ACM. Sept. 2007.
4. Lien, Richard, and Rolf W. Reitzig. Case Study: Realizing 40 Percent Software Development Productivity Improvements Using RUP, Agile, CMM/CMMI, and Organizational Transformation Methods. CMMI Technology Conference and User Group. Denver, CO., Nov. 2004.
5. Tsuda, M., Y. Morioka, and M. Takahashi. "Productivity Analysis of Software Development With an Integrated CASE Tool." Hitachi. 1990.
6. Agile Alliance. "Survey: The State of Agile Development." VersionOne. Aug. 2007.
7. Lim, W.C. "Effects of Reuse on Quality, Productivity, and Economics." IEEE Software Vol. 11, Issue 5. Sept. 1994: 23-30.
8. Middleton, Peter, and James Sutton. Lean Software Strategies: Proven Techniques for Managers and Developers. New York: Productivity Press, 2005.
9. McCormick, John W. "Software Engineering Education: On the Right Track." CROSSTALK. Aug. 2000 <www.stsc.hill.af.mil/crosstalk/2000/08/mccormick.html>.
10. Kennedy, Michael N. Product Development for the Lean Enterprise: Why Toyota's System Is Four Times More Productive and How You Can Implement It. Richmond, VA: Oaklea Press, 2003.
11. Womack, James P., Bruce M. Patton, and Daniel T. Jones. Lean Thinking. New York: Free Press, 2003.
12. Fisher, Roger, and William Ury. Getting to Yes: Negotiating Agreement Without Giving In. New York: Houghton Mifflin, 1992.
13. Brassard, Michael. The Memory Jogger Plus + Featuring the Seven Management and Planning Tools. Methuen, MA: GOAL/QPC, 1996.
14. Saaty, Thomas. Decision Making for Leaders. 3rd ed. Pittsburgh: RWS Publications, 2001.
15. Sauerwein, E., F. Bailom, Kurt Matzler, and Hans H. Hinterhuber. The Kano Model: How to Delight Your Customers. International Working Seminar on Production Economics Innsbruck, Austria. 19-23 Feb. 1996 <www.competence-site.de/dienstleistung.nsf/3397D512929D8241C1256AD8004B0027/$File/kano-model.pdf>.
16. Madey, Jan, and David L. Parnas. "Function Documents for Computer Systems." Science of Computer Programming 25.1 (1995).
17. Kirby, Jim. Rewriting Requirements for Design. Proc. of the International Conference on Software Engineering and Applications. Cambridge: MA. 4-6 Nov. 2002.
18. Alspaugh, Thomas. SCR-Style Requirements <www.ics.uci.edu/~alspaugh/software/SCR.html>.
19. The QFD Institute <www.qfdi.org>.

Notes

1. Lean terminology has occasionally been applied to software ideas, but usually with only a passing resemblance to the meanings used in classic industries. More importantly in each case the author has seen, the underlying framework of thought that ties those words and concepts together in typical Lean Production is missing.
2. Some figures required additional derivation based upon information in the sources. Details are available from the author upon request.
3. According to the IMVP, the Massachusetts Institute of Technology initiative that evaluated Lean adoption in the automotive industry beginning in the 1980s.
4. Generally, the same sources as for productivity. The full explanations are omitted here for space. Details are available from the author upon request.
5. Again, allowing for any derivations needed to present the data in this format.

About the Author

James M. Sutton is a Certified Professional Systems Engineer and a Principal Engineer at Lockheed-Martin (LM) Aeronautics in Fort Worth, Texas. His book on Lean software, "Lean Software Strategies," won the 2007 Shingo Prize. As the lead software product and technical-process architect on several LM programs, he began developing a Lean software approach. Sutton holds a Black Belt in Modern QFD, is a certified Theory of Inventive Problem Solving (TRIZ) practitioner, and has spoken and published for numerous conferences.

101 Academy DR
MZ 8557
Fort Worth, TX 76108-3800
Phone: (817) 935-4011
Fax: (817) 935-5228
E-mail: james.m.sutton @lmco.com