Presented at the 2011 ISPA/SCEA Joint Annual Conference and Training Workshop - www.iceaaonline.com

Estimation Challenges for 21st Century Software Systems

Barry Boehm, Brad Clark, Thomas Tan
USC Center for Systems and Software Engineering
University of Southern California
Los Angeles, CA, USA
boehm@usc.edu, brad@software-metrics.com,
thomast@usc.edu

Raymond Madachy
Department of Systems Engineering
Naval Postgraduate School
Monterey, CA, USA
rjmadach@nps.edu

Wilson Rosa
Information Technology Division
Air Force Cost Analysis Agency
Arlington, VA, USA
Wilson.Rosa@pentagon.af.mil

I. Introduction

Several future trends will present significant challenges for cost estimation of 21st century software systems. Prominent among these trends are:

1. Rapid change, emergent requirements, and evolutionary development
2. Net-centric systems of systems
3. Model-Driven and Non-Developmental Item (NDI)-intensive systems
4. Ultrahigh software system assurance
5. Legacy maintenance and Brownfield development
6. Agile and Lean/Kanban development.

This paper summarizes our ongoing US Government-sponsored analysis of these trends. We discuss the implications for cost estimation capabilities. Methods are shown to handle some of these effects, but many software variabilities remain uncovered.

II. Rapid Change, Emergent Requirements, and Evolutionary Development

21st century software systems will encounter increasingly rapid change in their objectives, constraints, and priorities. This change will be necessary due to increasingly rapid changes in their competitive threats, technology, organizations, leadership priorities, and environments. It is thus increasingly infeasible to provide precise size and cost estimates if the systems' requirements are emergent rather than pre-specifiable. This has led to increasing use of strategies such as incremental and evolutionary development, and to experiences with associated new sizing and costing phenomena such as the Incremental Development Productivity Decline. It also implies that measuring the system's size by counting the number of source lines of code (SLOC) in the delivered system may be an underestimate, as a good deal of software may be developed and deleted before delivery due to changing priorities.

There are three primary options for handling these sizing and estimation challenges. The first is to improve the ability to estimate requirements volatility during development via improved data collection and analysis, such as the use of code counters able to count numbers of SLOC added, modified, and deleted during development [1]. If such data is unavailable, the best one can do is to estimate ranges of requirements volatility.

For incremental and evolutionary development projects, the second option is to treat the earlier increments as reused software, and to apply reuse factors to them (such as the percent of the design, code, and integration modified, perhaps adjusted for degree of software understandability and programmer unfamiliarity [2]). This can be done either uniformly across the set of previous increments, of by having these factors vary by previous increment or by subsystem. This will produce an equivalent-SLOC (ESLOC) size for the effect of modifying the previous increments, to be added to the size of the new increment in estimating effort for the new increment. In tracking the size of the overall system, it is important to remember that these ESLOC are not actual lines of code to be included in the size of the next release.

The third option is to include an Incremental Development Productivity Decline (IDPD) factor, or even multiple factors varying by increment or subsystem. The IDPD factor is the percent of productivity decline between successive increments *i* and *i+1* such that $Productivity_{i+1} = (1 - IDPD\ \%) * Productivity_{i}$.

Unlike hardware, where unit costs tend to decrease with added production volume, the unit costs of later software increments tend to increase, due to previous-increment breakage and usage feedback, and due to increased integration and test effort. Thus, using hardware-driven or traditional software-driven estimation methods for later increments will lead to underestimates and overruns in both cost and schedule.

A relevant example was a large defense software system that had the following characteristics, where effort is measured in Person-Months (PM):

- 5 builds, 7 years, $100M
- Build 1 producibility over 300 SLOC/PM

- Build 5 producibility under 150 SLOC/PM
  - Including Build 1-4 breakage, integration, rework
  - 318% change in requirements across all builds.

A factor-of-2 decrease in producibility across four new builds corresponds to an average build-to-build IDPD factor of 19%. A recent quantitative IDPD analysis of a smaller software system yielded an IDPD of 14%, with significant variations from increment to increment [3]. Similar IDPD phenomena have been found for large commercial software such as the multi-year slippages in the delivery of Microsoft's Word for Windows [4] and Windows Vista, and for large agile-development projects that assumed a zero IDPD factor [5].

Based on experience with similar projects, the following impact causes and ranges per increment are conservatively stated in Table 1.

TABLE I. IDPD EFFORT DRIVERS

Less effort due to more experienced personnel, assuming reasonable initial experience level	
Variation depending on personnel turnover rates	5-20%
More effort due to code base growth	
Breakage, maintenance of full code base	20-40%
Diseconomies of scale in development, integration	10-25%
Requirements volatility, user requests	10-25%

In the best case, there would be 20% more effort (from above -20+20+10+10); for a 4-build system, the IDPD would be 6%. In the worst case, there would be 85% more effort (from above 40+25+25-5); for a 4-build system, the IDPD would be 23%.

With fixed staff size in any case, there would be either a schedule increase or incomplete builds. The difference between 6% and 23% may not look too serious, but the cumulative effect on schedule across a number of builds is very serious.

A simplified illustrative model relating productivity decline to number of builds needed to reach 4M ESLOC across 4 builds follows. Assume that the two year Build 1 production of 1M SLOC can be developed at 200 SLOC/PM, so it will need 208 developers (500 PM/ 24 mo.). Further assuming that constant staff size of 208 for all builds, the analysis in Figure 1 shows the impact on the amount of software delivered per build and the resulting effect on the overall delivery schedule as a function of the IDPD factor.

Many incremental development cost estimates assume an IDPD of zero, and an on-time delivery of 4M SLOC in 4 builds. However, as the IDPD factor increases and the staffing level remains constant, the productivity decline per build stretches the schedule out to twice as long for an IDPD of 20% as seen in Figure 1.

Thus, it is important to understand the IDPD factor and its influence when doing incremental or evolutionary development. Ongoing research indicates that the magnitude of the IDPD factor may vary by type of application (infrastructure software having higher IDPDs since it tends to be tightly coupled and touches everything; applications software having lower IDPDs if it is architected to be loosely coupled), or by recency of the build (older builds may be more stable). Further data collection and analysis would be very helpful in improving the understanding of the IDPD factor.

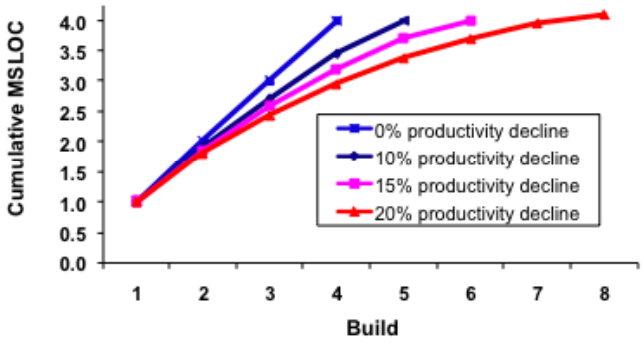

Figure 1. Effects of IDPD on Number of Builds to achieve 4M SLOC

III. 2. NET-CENTRIC SYSTEMS OF SYSTEMS (NCSOS)

If one is developing software components for use in a NCSoS, changes in the interfaces between the component systems and independently-evolving NCSoS-internal or NCSoS-external systems will add further effort. The amount of effort may vary by the tightness of the coupling among the systems; the complexity, dynamism, and compatibility of purpose of the independently-evolving systems; and the degree of control that the NCSoS protagonist has over the various component systems. The latter ranges from Directed SoS (strong control), through Acknowledged (partial control) and Collaborative (shared interests) SoSs, to Virtual SoSs (no guarantees) [6].

For estimation, one option is to use requirements volatility as a way to assess increased effort. Another is to use existing models such as COSYSMO [7] to estimate the added coordination effort across the NCSoS [8]. A third approach is to have separate models for estimating the systems engineering, NCSoS component systems development, and NCSoS component systems integration to estimate the added effort [7].

IV. 3. MODEL-DRIVEN AND NON-DEVELOPMENTAL ITEM (NDI)-INTENSIVE DEVELOPMENT

Model-driven development and Non-Developmental Item (NDI)-intensive development are two approaches that enable large portions of software-intensive systems to be generated from model directives or provided by NDIs such as commercial-off-the-shelf (COTS) components, open source components, and purchased services such as Cloud services. Figure 2 shows trends in the growth of COTS-based applications (CBAs) [10] in the top graph, and services-intensive systems [11] in the area of web-based e-services below it.

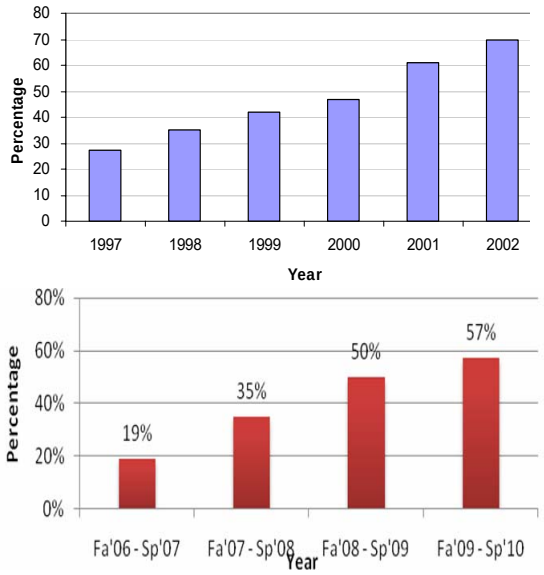

Figure 2. COTS and Services-Intensive Systems Growth in USC E-Services Projects

Such applications are highly cost-effective, but present several sizing and cost estimation challenges:

- Model directives generate source code in Java, C++, or other third-generation languages, but unless the generated SLOC are going to be used for system maintenance, their size as counted by code counters should not be used for development or maintenance cost estimation.
- Counting model directives is possible for some types of model-driven development, but presents significant challenges for others (e.g., GUI builders).
- Except for customer-furnished or open-source software that is expected to be modified, the size of NDI components should not be used for estimating.
- A significant challenge is to find appropriately effective size measures for such NDI components. One approach is to use the number and complexity of their interfaces with each other or with the software being developed. Another is to count the amount of glue-code SLOC being developed to integrate the NDI components, with the proviso that such glue code tends to be about 3 times as expensive per SLOC as regularly-developed code [12]. A similar approach is to use the interface elements of function points for sizing [13].
- A further challenge is that much of the effort in using NDI is expended in assessing candidate NDI components and in tailoring them to the given application. Some initial guidelines for estimating such effort are provided in the COCOTS model [14].
- Another challenge is that the effects of COTS and Cloud-services evolution are generally underestimated during software maintenance. COTS products generally provide significant new releases on the average of about every 10 months, and generally become unsupported after three new releases. With Cloud services, one does not have the option to decline new releases, and updates occur more frequently. One way to estimate this source of effort is to consider it as a form of requirements volatility.
- Another serious concern is that functional size measures such as function points, use cases, or requirements will be highly unreliable until it is known how much of the functionality is going to be provided by NDI components or Cloud services.

V. 4. Ultrahigh Software Systems Assurance

The increasing criticality of software to the safety of transportation vehicles, medical equipment, or financial resources; the security of private or confidential information; and the assurance of "24/7" Internet, web, or Cloud services will require further investments in the development and certification of software than are provided by most current software-intensive systems.

While it is widely held that ultrahigh-assurance software will substantially raise software–project cost, different models vary in estimating the added cost. For example, [15] estimates that engineering highly–secure software will increase costs by a factor of 8; the 1990's Softcost-R model estimates a factor of 3.43 [16]; the SEER model uses a similar value of 3.47 [13].

A recent experimental extension of the COCOMO II model called COSECMO used the 7 Evaluated Assurance Levels (EALs) in the ISO Standard Common Criteria for Information Technology Security Evaluation (CC) [ISO 1999], and quoted prices for certifying various EAL security levels to provide an initial estimation model in this context [18]. Its added-effort estimates were a function of both EAL level and software size: its multipliers for a 5000-SLOC secure system were 1.50 for EAL 4 and 8.8 for EAL 7.

A further sizing challenge for ultrahigh-assurance software is that it requires more functionality for such functions as security audit, communication, cryptographic support, data protection, etc. These may be furnished by NDI components or may need to be developed for special systems.

VI. 5. Legacy Maintenance and Brownfield Development

Fewer and fewer software-intensive systems have the luxury of starting with a clean sheet of paper or whiteboard on which to create a new Greenfield system. Most software-intensive systems are already in maintenance; The International Data Corporation estimates that there are roughly 200 billion SLOC in service worldwide [19]. Also, most new applications need to consider continuity of service from the legacy system(s) they are replacing. Many such applications involving incremental development have failed

because there was no way to separate out the incremental legacy system capabilities that were being replaced. Thus, such applications need to use a Brownfield development approach that concurrently architect the new version and its increments, while re-engineering the legacy software to accommodate the incremental phase-in of the new capabilities [20] [21] [22].

Traditional software maintenance sizing models have determined an equivalent SLOC size by multiplying the size of the legacy system by its Annual Change Traffic ACT) fraction (% of SLOC added + % of SLOC modified)/100. The resulting equivalent size is used to determine a nominal cost of a year of maintenance, which is then adjusted by maintenance-oriented effort multipliers. These are generally similar or the same as those for development, except for some, such as required reliability and degree of documentation, in which larger development investments will yield relative maintenance savings. Some models such as SEER [13] include further maintenance parameters such as personnel and environment differences. An excellent summary of software maintenance estimation is in [20].

However, as legacy systems become larger and larger (a full-up BMW contains roughly 100 million SLOC [24]), the ACT approach becomes less stable. The difference between an ACT of 1% and an ACT of 2% when applied to 100 million SLOC is 1 million SLOC. A recent revision of the COCOMO II software maintenance model sizes a new release as ESLOC = 2*(Modified SLOC) + Added SLOC + 0.5* (Deleted SLOC). The coefficients are rounded values determined from the analysis of data from 24 maintenance activities [Nguyen, 2010], in which the modified, added, and deleted SLOC were obtained from a code counting tool. This model can also be used to estimate the equivalent size of re-engineering legacy software in Brownfield software development. At first, the estimates of legacy SLOC modified, added, and deleted will be very rough, and can be refined as the design of the maintenance modifications or Brownfield re-engineering is determined.

VII. 6. Agile and Lean/Kanban Development

The difficulties of software maintenance estimation can often be mitigated by using lean workflow management techniques such as Kanban [25]. In Kanban, individual maintenance upgrades are given Kanban cards (Kanban is the Japanese word for card; the approach originated with the Toyota Production System). Workflow management is accomplished by limiting the number of cards introduced into the development process, and pulling the cards into the next stage of development (design, code, test, release) when open capacity is available (each stage has a limit of the number of cards it can be processing at a given time). Any buildups of upgrade queues waiting to be pulled forward are given management attention to find and fix bottleneck root causes or to rebalance the manpower devoted to each stage of development. A key Kanban principle is to minimize work in progress.

An advantage of Kanban is that if upgrade requests are relatively small and uniform, that there is no need to estimate their required effort; they are pulled through the stages as capacity is available, and if the capacities of the stages are well-tuned to the traffic, work gets done on schedule. However, if a too-large upgrade is introduced into the system, it is likely to introduce delays as it progresses through the stages. Thus, some form of estimation is necessary to determine right-size upgrade units, but it does not have to be precise as long as the workflow management pulls the upgrade through the stages. For familiar systems, performers will be able to right-size the units. For Kanban in less-familiar systems, and for sizing builds in agile methods such as Scrum, group consensus techniques such as Planning Poker [26] or Wideband Delphi [27] can generally serve this purpose.

The key point is to recognize that estimation of knowledge work can never be perfect, and to create development approaches that compensate for variations in estimation accuracy. Kanban is one such; another is the agile methods' approach of timeboxing or schedule-as-independent-variable (SAIV), in which maintenance upgrades or incremental development features are prioritized, and the increment architected to enable dropping of features to meet a fixed delivery date (With Kanban, prioritization occurs in determining which of a backlog of desired upgrade features gets the next card).

Such prioritization is a form of value-based software engineering, in that the higher-priority features can be flowed more rapidly through Kanban stages [24], or in general given more attention in defect detection and removal via value-based inspections or testing [28] [29]. Another important point is that the ability to compensate for rough estimates does not mean that data on project performance does not need to be collected and analyzed. It is even more important as a sound source of continuous improvement and change adaptability efforts.

VIII. 7. Putting It All Together at the Large-Project or Enterprise Level

The biggest challenge of all is that the six challenges above need to be addressed concurrently. Suboptimizing on individual-project agility runs the risks of easiest-first lock-in to unscalable or unsecurable systems, or of producing numerous incompatible stovepipe applications. Suboptimizing on security assurance and certification runs the risks of missing early-adopter market windows, of rapidly responding to competitive threats, or of creating inflexible, user-unfriendly systems.

One key strategy for addressing such estimation and performance challenges is to recognize that large systems and enterprises are composed of subsystems that have different need priorities and can be handled by different estimation and performance approaches. Real-time, safety-critical control systems and security kernels need high assurance, but are relatively stable. GUIs need rapid adaptability to change, but with GUI-builder systems, can largely compensate for lower assurance levels via rapid fixes. A key point is that for most enterprises and large

systems, there is no one-size-fits-all method of sizing, estimating, and performing.

This implies a need for guidance on what kind of process to use for what kind of system or subsystem, and on what kinds of sizing and estimation capabilities fit what kinds of processes. A start toward such guidance is provided in Table 2 [29].

TABLE II. SITUATION-DEPENDENT PROCESSES AND ESTIMATION APPROACHES

Type	Examples	Pros	Cons	Cost Estimation
Single Step	Stable; High Assurance	Pre-specifiable full-capability requirements	Emergent requirements or rapid change	Single-increment parametric estimation models
Prespecified Sequential	Platform base plus PPPIs	Pre-specifiable full-capability requirements	Emergent requirements or rapid change	COINCOMO or repeated single-increment parametric model estimation with IDPD
Evolutionary Sequential	Small: Agile Large: Evolutionary Development	Adaptability to change	Easiest-first; late, costly breakage	Small: Planning poker-type Large: Parametric with IDPD and Requirements Volatility
Evolutionary Overlapped	COTS-intensive systems	Immaturity risk avoidance	Delay may be non competitive	Parametric with IDPD and Requirements Volatility
Evolutionary Concurrent	Mainstream product lines; Systems of systems	High assurance with rapid change	Highly coupled systems with very rapid change	COINCOMO with IDPD for development; COSYSMO for rebaselining

Table 2 summarizes the traditional single-step waterfall process plus several forms of incremental development, each of which meets different competitive challenges and which are best served by different cost estimation approaches.

The time phasing of each form is expressed in terms of the increment 1, 2, 3, … content with respect to the Rational Unified Process (RUP) phases of Inception, Elaboration, Construction, and Transition (IECT) in Figure 3.

The profiles in Figure 3 are representative cost model outputs for these phases. They show estimated staffing levels over time using the COCOMO II default effort and schedule phase distributions for RUP [2]. However, the Evolutionary Concurrent process has the Inception and Elaboration phases for the next increment stretched to be coincident with the current increment.

It becomes complicated for estimation models to calculate effort and schedule per phase per iteration for all the cases in Figure 3. For example, in the incremental processes there are across-increment work dependencies to account for that become even more complex with overlapping phases.

The Single Step model is the traditional waterfall model, in which the requirements are pre-specified, and the system is developed to the requirements in a single increment. Single-increment parametric estimation models, complemented by expert judgment, are best for this process.

The Prespecified Sequential incremental development model is not evolutionary. It just splits up the development in order to field an early Initial Operational Capability, followed by several pre-planned product Improvements (P3Is). When requirements are identifiable and stable, it enables a strong, predictable process. When requirements are emergent and/or rapidly changing, it often requires very expensive rework when it needs to undo architectural commitments. Cost estimation can be performed by sequential application of single-step parametric models plus the use of an IDPD factor, or by parametric model extensions supporting the estimation of increments, including options for increment overlap and breakage of existing increments, such as the COINCOMO extension of COCOMO II.

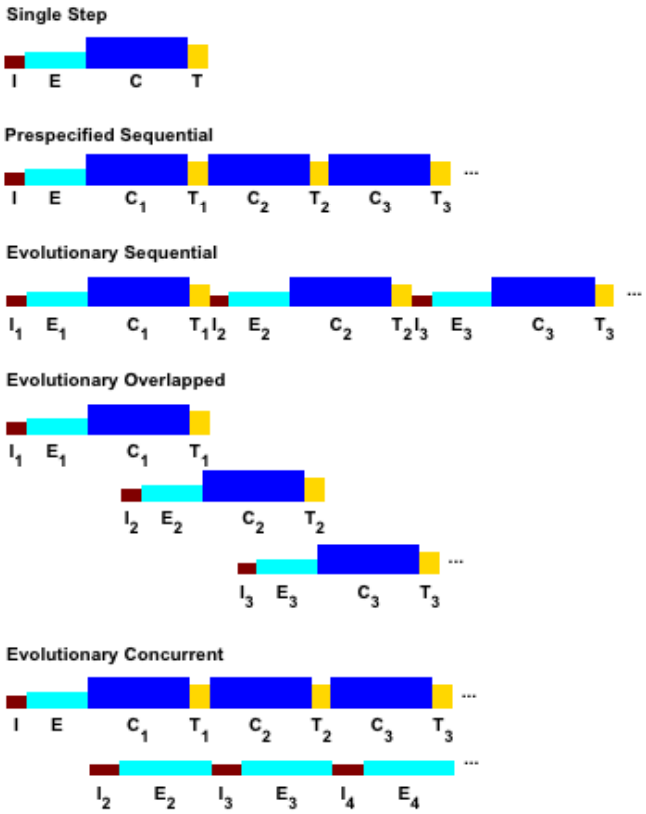

Figure 3. Lifecycle Process Phasing for Cost Estimation

The Evolutionary Sequential model rapidly develops an initial operational capability and upgrades it based on operational experience. Pure agile software development fits

this model: if something is wrong, it will be fixed in 30 days in the next release. Rapid fielding also fits this model for larger or hardware-software systems. Its strength is getting quick-response capabilities in the field. For pure agile, it can fall prey to an easiest-first set of architectural commitments which break when, for example, it tries to add security or scalability as a new feature in a later increment. For rapid fielding, it may be expensive to keep the development team together while waiting for usage feedback, but it may be worth it. For small agile projects, group consensus techniques such as Planning Poker are best; for larger projects, parametric models with an IDPD factor are best.

Evolutionary Overlapped covers the special case of deferring the next increment until critical enablers such as desired new technology, anticipated new commercial product capabilities, or needed funding becomes available or mature enough to be added.

Evolutionary Concurrent has the systems engineers handling the change traffic and rebaselining the plans and specifications for the next increment, while keeping the development stabilized for the current increment.

REFERENCES

[1] V. Nguyen, "Improved Size and Effort Estimation Models for Software Maintenance," PhD Dissertation, Department of Computer Science, University of Southern California, December 2010, http://csse.usc.edu/csse/TECHRPTS/by_author.html#Nguyen

[2] B. Boehm, C. Abts, W. Brown, S. Chulani, B. Clark, E. Horowitz, R. Madachy, D. Reifer, and B. Steece, Software Cost Estimation with COCOMO II, Upper Saddle River, NJ: Prentice-Hall, 2000.

[3] T. Tan, Q. Li, B. Boehm, Y. Yang, M. He, and R. Moazeni, "Productivity Trends in Incremental and Iterative Software Development," Proceedings, ACM-IEEE ESEM 2009.

[4] G. Gill, and M. Iansiti, “Microsoft Corporation: Office Business Unit,” Harvard Business School Case Study 691-033, 1994.

[5] A. Elssamadisy, and G. Schalliol, “Recognizing and Responding to ‘Bad Smells’ in Extreme Programming”, Proceedings, ICSE 2002, pp. 617-622.

[6] Systems Engineering Guide for System of Systems, Version 1.0, OUSD(AT&L), June 2008.

[7] R. Valerdi, The Constructive Systems Engineering Cost Model (COSYSMO), VDM Verlag, 2008.

[8] J. Lane, "Cost Model Extensions to Support Systems Engineering Cost Estimation for Complex Systems and Systems of Systems," 7th Annual Conference on Systems Engineering Research 2009 (CSER 2009)

[9] J. Lane, and B. Boehm, "Modern Tools to Support DoD Software Intensive System of Systems Cost Estimation - A DACS State-of-the-Art Report," August 2007.

[10] Y. Yang, J. Bhuta, B. Boehm, and D. Port, "Value-Based Processes for COTS-Based Applications," IEEE Software, Volume 22, Issue 4, July-August 2005, pp. 54-62.

[11] S. Koolmanojwong, and B. Boehm, "The Incremental Commitment Model Process Patterns for Rapid-Fielding Projects,"Proceedings, ICSP 2010, Paderborn, Germany.

[12] V. Basili and B. Boehm, "COTS-Based Systems Top 10 List," IEEE Computer, Vol. 34(5): 91-93, May, 2001.

[13] D. Galorath, and M. Evans, Software Sizing, Estimation, and Risk Management, Auerbach Publications, 2006.

[14] C. Abts, "Extending the COCOMO II Software Cost Model to Estimate Effort and Schedule for Software Systems Using Commercial-off-the-Shelf (COTS) Software Components: The COCOTS Model," PhD Dissertation, Department of Industrial and Systems Engineering, University of Southern California, May 2004.

[15] M. Bisignani, and T. Reed, “Software Security Costing Issues”, COCOMO Users' Group Meeting. 1988. Los Angeles: USC Center for Software Engineering.

[16] D. Reifer, Security: A Rating Concept for COCOMO II. 2002. Reifer Consultants, Inc.

[17] ISO JTC 1/SC 27, Evaluation Criteria for IT Security, in Part 1: Introduction and general model, International Organization for Standardization (ISO), 1999.

[18] E. Colbert, and B. Boehm, "Cost Estimation for Secure Software & Systems," ISPA / SCEA 2008 Joint International Conference, June 2008.

[19] H. Price, and J. Morley, “Create, Apply, and Amplify: A Story of Technology Development,” SEI Monitor, February 2009, page 2.

[20] R. Hopkins, and K. Jenkins, Eating the IT Elephant: Moving from Greenfield Development to Brownfield, IBM Press.

[21] G. Lewis, et al., “SMART: Analyzing the Reuse Potential of Legacy Components on a Service-Oriented Architecture Environment,” CMU/SEI-2008-TN-008.

[22] B. Boehm, “Applying the Incremental Commitment Model to Brownfield System Development,” Proceedings, CSER 2009.

[23] R. Stutzke, Estimating Software-Intensive Systems, Upper Saddle River, NJ: Addison Wesley, 2005.

[24] M. Broy, “Seamless Method- and Model-based Software and Systems Engineering,” The Future of Software Engineering, Springer, 2010.

[25] D.Anderson, Kanban, Blue Hole Press, 2010.

[26] M. Cohn, Agile Estimating and Planning, Prentice Hall, 2005.

[27] B. Boehm, Software Engineering Economics. Englewood Cliffs, NJ, Prentice-Hall, 1981.

[28] K. Lee, and B. Boehm, "Empirical Results from an Experiment on Value-Based Review (VBR) Processes," Proceedings, ISESE 2005, September 2005.

[29] Q. Li, B. Boehm, Y. Yang, and Q. Wang, "A Value-Based Review Process for Prioritizing Artifacts", Proceedings, ICSSP 2010.

[30] B. Boehm, and J. Lane, “"DoD Systems Engineering and Management Implications for Evolutionary Acquisition of Major Defense Systems," Proceedings, CSER 2010.